Reactive plan execution in multi-agent environments

César Augusto Guzmán Alvarez
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

A thesis submitted for the degree of
Título de Doctor por la Universitat Politècnica de València
Under the supervision of:
Prof. Eva Onaindía de la Rivaherrera

March 2019

Title:	Reactive plan execution in multi-agent environments
Author:	César Augusto Guzmán Alvarez
Advisors:	Prof. Eva Onaindía de la Rivaherrera

Reviewers:

Prof. Costin Badica
Prof. Jose Marcos Moreno Vega
Prof. Michele Malgeri

Day of the defense: April 5th, 2019

To my family.

Without you dedication and support, this thesis would not be possible.

Acknowledgements

Valencia, Spain César Augusto Guzmán Alvarez

December, 2018

To my family, my partner, my son, my mother, my father, and my brothers; Mairi, Martín, Esperanza, Carlos, y Carlos y Andres: because I owe it all to all of you. Many Thanks!

My sincere gratitude to my advisor Prof. Eva Onaindia for the continuous support of my Ph.D. study and related research, for accepting nothing less than excellence from me, for her patience, motivation, and immense knowledge. Her supervision helped me in all the time of this Ph.D. dissertation. She is the best advisor and mentor.

Besides my advisor, I would like to thank the rest of people who were directly or indirectly related to my Ph.D. dissertation. Some of them are Laura Sebastia, Inma Garcia, and Mario Rodriguez. Thanks for their insightful comments and encouragement, but also for the hard questions, which motivated me to widen my research from various perspectives.

My sincere thanks also go to Dr. Jeremy Frank and Dr. Alfredo Bencomo, who provided me an opportunity to join their team as an intern at NASA Ames Research Center, and gave me access to the research facilities. Without their precious support, it would not have been possible to start with this research. Also I thank my other friends at NASA institution. It was fantastic to have the opportunity to work the start point of my research in your facilities. What a cracking place to work!

And finally, last but by no means least, also I would like to thank everyone at the DSIC lab, it was great to share the laboratory with all of you.

Thanks for all your encouragement!

Abstract

One of the challenges of robotics is to develop control systems capable of quickly obtaining intelligent, suitable responses for the regularly changing that take place in dynamic environments. This response should be offered at runtime with the aim of resume the plan execution whenever a failure occurs. Automated classical planning works on deliberative tools to calculate plans that achieve operation goals. The term *reactive planning* addresses all the mechanisms that, directly or indirectly, promote the resolution of failures during the plan execution. Reactive planning systems work under a continual planning and execution approach, i.e., interleaving planning and execution in dynamic environments.

Most of the current research puts the focus on developing reactive planning system that works on single-agent scenarios to recover quickly plan failures, but, if this is not possible, we may require more complex multi-agent architectures where several agents may participate to solve the failures. Therefore, continual planning and execution systems have usually conceived solutions for individual agents. The complexity of establishing agent communications in dynamic and time-restricted environments has discouraged researchers from implementing multi-agent collaborative reactive solutions.

In line with this research, this Ph.D. dissertation attempts to overcome this gap and presents a multi-agent reactive planning and execution model that keeps track of the execution of an agent to recover from incoming failures.

Firstly, we propose an architecture that comprises a general reactive planning and execution model that endows a single-agent with monitoring and execution capabilities. The model also comprises a reactive planner module that provides the agent with fast responsiveness to recover from plan failures. Thus, the mission of an execution agent is to monitor, execute and repair a plan, if a failure occurs during the plan execution.

The reactive planner builds on a time-bounded search process that seeks a recovery plan in a solution space that encodes potential fixes for a failure. The agent generates the search space at runtime with an iterative time-bounded construction that guarantees that a solution space will always be available for attending an immediate plan failure. Thus, the only operation that needs to be done when a failure occurs is to search over the solution space until a recovery path is found. We evaluated the performance and reactiveness of our single-agent reactive planner by conducting two experiments. We have evaluated the reactiveness of the single-agent reactive planner when building solution spaces within a given time limit as well as the performance and quality of the found solutions when compared with two deliberative planning methods.

Following the investigations for the single-agent scenario, our proposal is to extend the single model to a multi-agent context for collaborative repair where at least two agents participate in the final solution. The aim is to come up with a multi-agent reactive planning and execution model that ensures the continuous and uninterruptedly flow of the execution agents. The multi-agent reactive model provides a collaborative mechanism for repairing a task when an agent is not able to repair the failure by itself. It exploits the reactive planning capabilities of the agents at runtime to come up with a solution in which two agents participate together, thus preventing agents from having to resort to a deliberative solution. Throughout the thesis document, we motivate the application of the proposed model to the control of autonomous space vehicles in a Planetary Mars scenario.

To evaluate our system, we designed different problem situations from three real-world planning domains. We tested the reactiveness and performance of our approach along a complete execution of the tasks from the planning domains, demonstrating that our model is a very suitable multi-agent mechanism to fix failures that represent slight deviations from the main course of the plan.

Finally, the document presents some conclusions and also outlines future research directions.

Resumen

Uno de los desafíos de la robótica es desarrollar sistemas de control capaces de obtener rápidamente respuestas adecuadas e inteligentes para los cambios constantes que tienen lugar en entornos dinámicos. Esta respuesta debe ofrecerse al momento con el objetivo de reanudar la ejecución del plan siempre que se produzca un fallo en el mismo. La planificación clásica automática trabaja con herramientas deliberativas para calcular planes que consigan los objetivos. El término *planificación reactiva* aborda todos los mecanismos que, directa o indirectamente, promueven la resolución de fallos durante la ejecución del plan. Los sistemas de planificación reactiva funcionan bajo un enfoque de planificación y ejecución continua, es decir, se intercala planificación y ejecución en entornos dinámicos.

Muchas de las investigaciones actuales se centran en desarrollar planificadores reactivos que trabajan en escenarios de un único agente para recuperarse rápidamente de los fallos producidos durante la ejecución del plan, pero, si esto no es posible, pueden requerirse arquitecturas de múltiples agentes y métodos de recuperación más complejos donde varios agentes puedan participar para solucionar el fallo. Por lo tanto, los sistemas de planificación y ejecución continua generalmente generan soluciones para un solo agente. La complejidad de establecer comunicaciones entre los agentes en entornos dinámicos y con restricciones de tiempo ha desanimado a los investigadores a implementar soluciones reactivas donde colaboren varios agentes.

En línea con esta investigación, la presente tesis doctoral intenta superar esta brecha y presenta un modelo de ejecución y planificación reactiva multiagente que realiza un seguimiento de la ejecución de un agente para reparar los fallos con ayuda de otros agentes.

En primer lugar, proponemos una arquitectura que comprende un modelo general reactivo de planificación y ejecución que otorga a un agente capacidades de monitorización y ejecución. El modelo también incorpora un planificador reactivo que proporciona al agente respuestas rápidas para recuperarse de los fallos que se pueden producir durante la ejecución del plan. Por lo tanto, la misión de un agente de ejecución es monitorizar, ejecutar y reparar un plan, si ocurre un fallo durante su ejecución.

El planificador reactivo está construido sobre un proceso de busqueda limitada en el tiempo que busca soluciones de recuperación para posibles fallos que pueden ocurrir. El agente genera los espacios de búsqueda en tiempo de ejecución con una construcción iterativa limitada en el tiempo que garantiza que el modelo siempre tendrá un espacio de búsqueda disponible para atender un fallo inmediato del plan. Por lo tanto, la única operación que debe hacerse es buscar en el espacio de búsqueda hasta que se encuentre una solución de recuperación. Evaluamos el rendimiento y la reactividad de nuestro planificador reactivo mediante la realización de dos experimentos. Evaluamos la reactividad del planificador para construir espacios de búsqueda dentro de un tiempo disponible dado, asi como támbien, evaluamos el rendimiento y calidad de encontrar soluciones con otros dos métodos deliberativos de planificación.

Luego de las investigaciones de un solo agente, propusimos extender el modelo a un contexto de múltiples agentes para la reparación colaborativa donde al menos dos agentes participan en la solución final. El objetivo era idear un modelo de ejecución y planificación reactiva multiagente que garantice el flujo continuo e ininterrumpido de los agentes de ejecución. El modelo reactivo multiagente proporciona un mecanismo de colaboración para reparar una tarea cuando un agente no puede reparar la falla por sí mismo. Explota las capacidades de planificación reactiva de los agentes en tiempo de ejecución para encontrar una solución en la que dos agentes participan juntos, evitando así que los agentes tengan que recurrir a mecanismos deliberativos. Durante todo el documento de la tesis, motivamos la aplicación del modelo propuesto para el control de vehículos espaciales autónomos, escenario de los robots de Marte.

Para evaluar nuestro sistema, diseñamos diferentes situaciones en tres dominios de planificación del mundo real. Probamos la reactividad y el rendimiento de nuestro enfoque a lo largo de una ejecución completa de las tareas de los dominios de planificación, demostrando que nuestro modelo es un mecanismo multiagente adecuado para reparar fallos que representan una pequeña desviación en la ejecución del plan.

Finalmente, el documento presenta algunas conclusiones y también propone futuras líneas de investigación posibles.

Resum

Un dels desafiaments de la robòtica és desenvolupar sistemes de control capaços d'obtindre ràpidament respostes adequades i intel·ligents per als canvis constants que tenen lloc en entorns dinàmics. Aquesta resposta ha d'oferir-se al moment amb l'objectiu de reprendre l'execució del pla sempre que es produïsca una fallada en aquest. La planificació clàssica automàtica treballa amb eines deliberatives per a calcular plans que aconseguisquen els objectius. El terme *planificació reactiva* aborda tots els mecanismes que, directa o indirectament, promouen la resolució de fallades durant l'execució del pla. Els sistemes de planificació reactiva funcionen sota un enfocament de planificació i execució contínua, és a dir, s'intercala planificació i execució en entorns dinàmics.

Moltes de les investigacions actuals se centren en desenvolupar planificadors reactius que treballen en escenaris d'un únic agent per a recuperar-se ràpidament de les fallades produïdes durant l'execució del pla, però, si això no és possible, poden requerir-se arquitectures de múltiples agents i mètodes de recuperació més complexos on diversos agents puguen participar per a solucionar la fallada. Per tant, els sistemes de planificació i execució contínua generalment generen solucions per a un sol agent. La complexitat d'establir comunicacions entre els agents en entorns dinàmics i amb restriccions de temps ha desanimat als investigadors a implementar solucions reactives on col·laboren diversos agents.

En línia amb aquesta investigació, la present tesi doctoral intenta superar aquesta bretxa i presenta un model d'execució i planificació reactiva multiagent que realitza un seguiment de l'execució d'un agent per a reparar les fallades amb ajuda d'altres agents.

En primer lloc, proposem una arquitectura que comprén un model general reactiu de planificació i execució que atorga a un agent capacitats de monitoratge i execució. El model també incorpora un planificador reactiu que proporciona a l'agent respostes ràpides per a recuperar-se de les fallades que es poden produir durant l'execució del pla. Per tant, la missió d'un agent d'execució és monitorar, executar i reparar un pla, si ocorre una fallada durant la seua execució.

El planificador reactiu està construït sobre un procés de cerca limitada en el temps que busca solucions de recuperació per a possibles fallades que poden ocórrer. L'agent genera els espais de cerca en temps d'execució amb una construcció iterativa limitada en el temps que garanteix que el model sempre tindrà un espai de cerca disponible per a atendre una fallada immediata del pla. Per tant, l'única operació que ha de fer-se és buscar en l'espai de cerca fins que es trobe una solució de recuperació. Avaluem el rendiment i la reactivitat del nostre planificador reactiu mitjançant la realització de dos experiments. Avaluem la reactivitat del planificador per a construir espais de cerca dins d'un temps disponible donat, així com també, avaluem el rendiment i qualitat de trobar solucions amb altres dos mètodes deliberatius de planificació.

Després de les investigacions d'un sol agent, vam proposar estendre el model a un context de múltiples agents per a la reparació col·laborativa on almenys dos agents participen en la solució final. L'objectiu era idear un model d'execució i planificació reactiva multiagent que garantisca el flux continu i ininterromput dels agents d'execució. El model reactiu multiagent proporciona un mecanisme de col·laboració per a reparar una tasca quan un agent no pot reparar la falla per si mateix. Explota les capacitats de planificació reactiva dels agents en temps d'execució per a trobar una solució en la qual dos agents participen junts, evitant així que els agents hagen de recórrer a mecanismes deliberatius. Durant tot el document de la tesi, motivem l'aplicació del model proposat per al control de vehicles espacials autònoms, escenari dels robots de Mart.

Per a avaluar el nostre sistema, dissenyem diferents situacions en tres dominis de planificació del món real. Provem la reactivitat i el rendiment del nostre enfocament al llarg d'una execució completa de les tasques dels dominis de planificació, demostrant que el nostre model és un mecanisme multiagent adequat per a reparar fallades que representen una xicoteta desviació en l'execució del pla.

Finalment, el document presenta algunes conclusions i també proposa futures línies d'investigació possibles.

Contents

Glossary

AI Artificial Intelligence. 1, 6

CP Centralize Planner. 178, 188, 192, 193, 196, 198, 199

CoDMAP Competition of Distributed and Multi agent Planners. 165

DP Deliberative Planner. 29

EX EXecution. 29

FMAP Forward Multi-Agent Planning. 38

ICAPS International Conference on Automated Planning and Scheduling. 165

IPC International Planning Competition. 83, 101, 165–168, 188, 222, 248

MA-PDDL Multi-Agent Planning Domain Definition Language. 165

MAP Multi-Agent Planning. 2, 28, 31, 37, 38, 164, 165, 168–170

MARPE Multi-Agent Reactive Planning and Execution. 121, 122, 124, 125, 127, 150, 152, 153, 157–160, 162, 164, 167–169, 176, 184, 200, 208, 209, 254

MAS Multi-Agent System. 19–21, 23, 25

MD Multi-Deliberative repair. 160, 161, 163, 169–171, 173, 174, 178–182, 184, 185, 188, 191–199

MO MOnitoring. 29

MR Multi-Reactive repair. 160–163, 169–173, 178–188, 191–199

PDDL Planning Domain Definition Language. 54–56, 59, 61, 80, 133, 248

RPE Reactive Planning and Execution. 48–50, 52, 61, 62, 74, 79, 124, 130, 131, 159

RP Reactive Planner. 29, 156, 179, 182, 185, 191, 197

SD Single-Deliberative repair. 160–163, 169–171, 179, 180, 182

SR Single-Reactive repair. 160–163, 169–174, 178–182, 184–188, 191–193, 196–199

List of definitions

List of Tables

List of Figures

List of Algorithms

Chapter 1

Motivation

"Everything is within your power, and your power is within you."

(Janice Trachtman)

Much of the research on *Artificial Intelligence* (AI) planning is aimed at generating domain-independent planning technology [42] in a great variety of industries applications such as manufacturing [68], route planning [99, 21], military and civilian coalition operations [83], space exploration [19], and so forth. Inside the AI planning community, the application of planning to industry gives rise to special-purpose systems, which are expensive to extend to other cases, where automated planning and plan execution need to be integrated. While the primary focus of automated planning is on deliberative tools to calculate plans that achieve operation goals, the focus of plan execution is on developing control methods over relatively short time spans to ensure the plan actions are executed stably [10]. The term reactive planning [75] has been used to address all the mechanisms that, directly or indirectly, promote the resolution of failures in planning and execution. planning and execution are systems that work under a continual planning approach [16], i.e. interleaving planning and execution in a world under continual change.

Reactive planning is proposed as one key mechanism in fast failure resolution. Reactive planning differs from classical planning in two aspects. First, it operates in a timely fashion and hence is extremely important with highly dynamic and unpredictable environments. Second, it computes just one next action in every instant, based on the current context. The Reactive planners often (but not always) exploit reactive plans, which are stored in complex structures. Classically, reactive planning has been approached from different perspectives [18]: 1) responding very quickly to changes in the environment through a reactive plan library that stores the best course of action to each possible contingency, 2) choosing the immediate next action on the basis of the current context, and 3) using more complex structures in order to handle execution failures or environmental changes. The approach 1), used by early planning and execution systems [47], implies storing a plan for each possible state of the world, an option which is not affordable in highly dynamic environments. The hierarchical control structures is the approach followed by the models that emphasize reactiveness, computing just one next action in every instant based on the current context [30, 72]. They provide, to an executor agent, a deliberative mechanism to choose the immediate next action when a quick response is required in unpredictable environments [17].

Multi-Agent Planning (MAP) systems are extensions of planning and execution single-agent frameworks for distributed problem solving [113, 108, 65]. One common characteristic of MAP architectures is that the multi-agent infrastructure is specifically used for supporting the deliberative machinery (planning) whereas the need for reactive mechanisms (plan execution) is basically relegated to the individual agent level. Thus, when an executor agent encounters a failure during plan execution it must be capable of repairing it by either chooses the immediate next action, or by having task assessors that abstractly plan how to accomplish the failed task [65], or by consulting a plan library of predefined, static plans [98] (previous explained approaches 1, 2, and 3).

Despite the fact that reactive planning has been studied since the 90's decade, there is still a wide range of problems whose solution has not been treated in the literature.

On the one hand, nowadays planning control applications need to rely on a robust plan execution capable to return a reactive response that repairs the non-executable part of the plan and allows the flow of the execution to engage with the rest of the (executable) plan. Under these premises, reactive planning as the utilization of a library of reactive plans that accounts for all possible contingencies in the world is unaffordable; and abstract repair plans are not sufficient nor executable. Other approaches resort to reactive model-based programming languages to combine the flexibility of reactive execution languages and the reasoning power of deliberative planners, and thus automate the process of reasoning about low-level system interactions [114, 32]. However, these approaches require the programmer specifically design the plans and contingencies that will ensure a high degree of success in each particular application. We argue that most reactive planning approaches, even though they care about fast resolution with completed plan library structures, have never exploited the idea of providing quick deliberative responses by using more complex structures that consider more deliberative (long horizon) responses rather than short-term reactiveness. In practice, they react to a change or failure but they do not guarantee a response within a limited time. Hence, they have not focused on the particularities of operate over bounded-size structures which have been generated within a limited time. The design of new computational approaches for reactive planning in bounded-size structures may lead to provide runtime deliberative responses in any domain applications.

On the other hand, most reactive planners have focused on single-agent scenarios where agents are capable of repairing their own plan failures, but, if this is not possible, more complex multi-agent scenarios may be required. For instance, in the Mars planning domain, which stems from a real-world problem that NASA is dealing with in some research projects on space exploration [67, 94], a rover that works on the Martian surface may lose capabilities [102, 64] (e.g. reduced mobility due to wear and tear on motors or mechanical systems, reduced power generation due to battery degradation, accumulated dust on solar arrays), as happened with the

Spirit and Opportunity rovers. If there is a plan failure and the rover is not able to repair it by its own repairing mechanism, the rover, before communicating to Earth (the communication has a long delay of at least 2.5 minutes, and at most 22 minutes), may cooperate with other rover or spacecraft in order to perform a joint repair. This property grants a high degree of reactivity and robustness as it always attempts first to repair failures at runtime, either individually or collectively, before resorting to replanning, i.e., asking the deliberative planner for a new executable plan. Therefore, we consider it to be of extreme importance.

Hence, this thesis pursues computational solutions for both complex scenarios: recovering plan failures by self-reactive planning techniques during execution, and solving plan failures with a collaborative approach between agents when the agent is not able to repair the plan by himself. This thesis has been developed under the umbrella of several research projects in domain-independent frameworks for single and multi-agent systems. Reactive planning and execution, more specifically, the research carried out in this thesis plays a significant role in those research projects. This thesis is developed under the framework of the following research projects funded by the Spanish Government:

- "PELEA: A domain-independent framework for Planning, replanning, monitoring, Execution, and LEArning" under grant TIN2008-06701-C03-01 (Main Researcher: Eva Onaindia, from 2008 to 2011). PELEA includes components that allow the agent to dynamically integrate deliberative planning and replanning, execution, monitoring and learning techniques. In full reactive applications, the learning and deliberative planning and replanning components might not be needed or can be used very rarely to set up the initial plan to carry out. The work of this thesis aims to provide reactive planning and execution mechanisms for a PELEA agent.

- "Multi-agent PlanInteraction" under grant TIN2011-27652-C03-01-AR (Main Researcher: Eva Onaindia, from 2011). In this project, we aim to analyze processes where groups of agents aim to cooperate to solve plan failures while considering their own interests. Agents share common information at the recovery process. However, each agent may have different personal situations that are possibly in conflict. Agent-based repairing techniques are a subset of the scenarios studied by the Multi-agent PlanInteraction project.

Additionally, the work of this thesis could have not been possible without the predoctoral fellowship program FPI (BES-2009-013327) and the short stay (EEBB-I12-04550) made at NASA for the same FPI program granted by the Spanish Government.

1.1 Objectives

The general objective of this Ph.D. dissertation is to develop a new Multi-Agent Reactive Planning and Execution model that includes a multi-agent reactive technique to recover from plan failures with the help of other agents. Specifically, we deal with an unexplored point of view, which consists of solving plan failures with a collaborative repair approach between agents. It is reasonable to assume that the agents need to share some information to collaborate with other agents. For that purpose, we propose the following specific objectives:

1. **State-of-the-art in reactive planning and execution:** It is necessary to survey, classify, and review the existing literature on automated reactive planning and related topics.
   (a) Review related architectures for single-agent and multiple-agents.
   (b) Examine the concept of reactive techniques with single-agent and multiple-agents.
   (c) Analyze the shortcomings of the techniques proposed in the literature for reactive planning and execution.

2. **Single-agent reactive planning and execution model:** In this objective, we develop a domain-independent reactive model which requires special features due to the limited resources of the executor agents that are usually employed. The model should include a reactive planner. Even though reactive planners in AI care about fast resolution, they have not focused on providing quick responses with complex structures that the agent generates at runtime within a limited time. Therefore, it is necessary to propose and validate a computational model for a single-agent reactive planning and execution.

   (a) Propose a general domain-independent reactive planning and execution model.

   (b) Propose a general reactive planning technique that employs reactive structures to recover from plan failures in a fashion response.

   (c) Validate the computational efficiency of the proposed mechanism, and compare it with both a general and well-known deliberative planner (LAMA planner [89]), and a plan-adaptation mechanism.

3. **Multi-agent reactive planning and execution model:** As far as we are concerned, the topic of multi-agent reactive planning and execution is introduced in automated reactive planning with this Ph.D. dissertation. The multi-agent repair is a complex process since the agent should solve the plan failure with the other agents, as fast as possible, before resulting to a replanning mechanism. Therefore, due to its novelty, we put a particular emphasis on exploring this type of complex solution to solve failures with a collaborative approach. We aim to propose a general multi-agent computational model for recovery from plan failures and analyze the impact of the conditions of the environment on agents performance.

   (a) Identify and analyze the workflow of information necessary that may help agents to perform successfully in multi-agent recovery process.

(b) Propose a general protocol and define information to share.

(c) Validate the optimal performance when the multi-reactive model is enabled and compare the computational efficiency of the multi-reactive model with other centralized systems.

1.2 Contributions

Our main contribution is a repair mechanism that allows execution agents to reactively and collaboratively attend a plan failure during execution. Specifically, it is a multi-agent reactive repair planner that employs bounded-structures to respond in a timely fashion to a somewhat dynamic and unpredictable environment. The multi-agent reactive planner allows execution agents to perform a general model, which enables a group of two agents to act coherently, overcoming the uncertainties of complex, dynamic environments to repair failures or inconsistent views of the world state. After defining a set of incidences of breakdowns for the different domains, this approach will be compared with some deliberative planner to study how fast the reactive planner generates a solution.

1.3 Document structure

The thesis is divided into five main parts. First, some theory behind planning and execution architectures, reactive planning, recovery and diagnosis, plan coordination, and multi-agent plan execution are discussed. Second, the single-agent reactive planning and execution model is presented and some test and evaluations are discussed. Next, the single-agent model is extended to a multi-agent context and some evaluations of this multi-agent model are presented and discussed. Finally, we discuss some conclusions and present some future works.

Chapter 2

State of the art in planning and execution

"Study the past if you would define the future."

(Confucius)

This chapter summarizes the literature on the principal research works related to the topics addressed in this Ph.D. dissertation. Particularly, we divide the state-of-the-art presentation in two main blocks:

1. **Planning and execution architectures**. The underlying framework of the model presented in this Ph.D. is a multi-agent planning and execution architecture which was developed within two research projects: PELEA, devoted to the design of a single-agent planning and execution architecture, and PlanInteraction, an extension of PELEA to a multi-agent environment. The first part of this document will cover the most significant planning and execution architectures, starting from the basic or single-agent architectures and then presenting some multi-agent system architectures.

2. **Approaches to planning and execution**. The ultimate objective of our planning and execution model is that agents cooperatively repair a failure in one of the agents' plan, if possible. We will revise in this subsection the most relevant single-agent approaches to plan repair and replanning as well as presenting some discussion on how they cam be extended to a multi-agent environment. Unlike deliberative planning, planners that calculate a plan at execution time must return a timely and promptly response. This area of research is commonly referred as to *reactive planning and execution*. The second block of this chapter is devoted to revise the main approaches of reactive planning and execution found in the literature for single and multi agent environments.

2.1 Planning and execution architectures

Planning and execution systems work under a *continual planning approach* [16], interleaving planning and execution in a dynamic world with continual change. Most of the existing approaches employ an architecture where an executor agent has a deliberative mechanism that allows the agent to choose the next immediate action during the execution of a plan in an unpredictable environment [17]. In other words, using a plan library, they calculate the next action at every time instant based on the current context [72]. Other planning and execution systems, however, design an architecture that integrates Planning, Execution and Monitoring inside the executor agent.

In this section, we will survey the most relevant architectures that interleave planning and execution in single-agent and multi-agent systems.

2.1.1 Single-agent frameworks

One of the main challenges addressed by the planning and execution community is the problem to design the framework that integrates Planning, Execution and

Monitoring inside an executor agent. Using expressive frameworks to implement an executor agent with these three behaviors is one of the key aspects of an efficient planning and execution system. In this section, we will study the most relevant frameworks that interleave planning and execution in single-agent systems.

2.1.1.1 Task Control Architecture

The Task Control Architecture (TCA) [101] is one of the first planning and execution architectures that has widely influenced most of the planning and execution works since the 90's. A robot agent built with TCA consists of task-specific modules, and a general purpose, reusable central control module. The task-specific modules process all robot-dependent information while the central control module is responsible for supervising and routing messages between the task-specific modules as well as maintaining the information of control. More concretely, TCA is a distributed architecture with a centralized control, which is the main limitation of this architecture because the central process may become a bottleneck. However, TCA offers facilities to implement the modules with different programming languages and puts the emphasis principally on the execution, monitoring and exception handling.

2.1.1.2 Distributed Architecture for Mobile Navigation

The Distributed Architecture for Mobile Navigation (DAMN) [92] is another architecture developed under the same concept as TCA. In DAMN, multiple modules concurrently share the control of the agent's devices. The modules vote for actions that satisfy their goals and asynchronously communicate the results to the central module at a rate adequate for the particular selection and execution of the actions. The central module, with no regard to the level of planning involved by each module, selects the appropriate actions based on the modules' votes and execute them. There can also be many independent central modules, each one responsible for the control of a different device. The independent central modules can run in parallel and communicate with each other, if necessary, for flexible coordination. In such a case, a central high-level module is required to control the information from the independent central modules.

2.1.1.3 BEhavior-based Robot Research Architecture

BEhavior-based Robot Research Architecture (BERRA) [66] is a three-layer architecture: the deliberative layer, which works as a deliberative planner; the task execution layer, which performs the monitoring process, verifying the correct execution of the action; and the reactive layer that is an executor of actions. The deliberative layer has two modules, the human-robot interface (HRI) and the planner; the task execution layer has also two modules, the task execution supervisor and the localizer modules; and the reactive layer contains three modules, the behaviors, the resources and the controllers modules. More specifically, the modules of each layer are responsible of:

1. **Deliberative layer**:
    - The HRI understands gesture and speech and also has a voice synthesizer for feedback.
    - The planner decomposes commands from the HRI in states.
2. **Task execution layer**:
    - The task execution supervisor module (TEM) receives information from the planner and translates it into a configuration of reactive components. It also informs the controllers which behaviors should send data to them.
    - The localizer tracks the robot position.
3. **Reactive layer**:

- The behaviors control the sensors and actuators and can accept connections from the controllers. They receive data from the TEM and inform this module, if necessary, about the success or failure of an action.
- Resources are for sharing sensor data. Clients of resources are behaviors or other resources. Clients must establish a connection with the resource to obtain information from a resource.
- The controllers are in charge of sending the actions to the actuators and receive the results data from the behaviors. Data coming from behaviors are used to produce the control signal to be sent to the actuators.

The BERRA architecture contains modules for deliberative planning, executing actions and monitoring but it does not accommodate a reactive planning module.

2.1.1.4 Tripodal schematic control architecture

The tripodal schematic control architecture [60] is also three-layer architecture like BERRA. The three layers with their functionalities are:

- **Deliberative layer**: the main tasks of the deliberative layer are to provide an interface with the user and to execute the centralized planning mechanism. It works in a high-level configuration.
- **Sequencing layer**: this layer consists of asynchronous and nonrealtime components that are classified into two groups. The first group is the process supervisor and configuration of low-level tasks. The second group is the information part that implement complex algorithms to extract data from sensors. Communication between components can be done synchronously or asynchronously.
- **Reactive layer**: the reactive layer handles hardware and real-time components. It works as a coordinator that decides in real-time the low-level command to execute in the hardware.

The main problem of the architecture is that it lacks a component that allows recovering the plan failure whenever it occurs. The reactive layer works only translating and executing high-level tasks to low-level tasks. Thus, when a plan failure occurs it goes to replanning by calling the deliberative layer. Another problem is that the deliberative planner is inside the executor agent which in many cases is unaffordable because the deliberative planner can take too much time to generate a plan for execution. Moreover, in this architecture, the deliberative planner is a centralized planning mechanism meaning that the planning process computes the solution plan of the agent from a global point of view.

2.1.1.5 Teleo-Reactive EXecutive architecture

The Teleo-Reactive EXecutive (T-REX) [71] combines planning and state estimation techniques within a hybrid executor. It integrates primitive robot actions into higher level tasks. An executor agent works as a coordinator of a set of modules that encapsulates all details of how to accomplish their objectives. The executor agent has three different modules:

1. The mission manager generates necessary actions automatically by using a planner.
2. The navigator and science operator manager translate high-level directives into executable commands depending on the current system state.
3. The executive manager encapsulates access to commands and state variables sensing from the environment.

T-REX represents and reasons about plans with a constraint-based temporal planning paradigm. A timeline represents all the states in the past, present, and future. This is an interesting feature though not particularly valuable when a reactive response is required and attention must be put on the next action to be executed.

T-REX provides not only a robust and safe execution but also high-level programming capabilities and goal-directed commanding through its onboard planner.

2.1.1.6 PELEA: single-agent planning and execution architecture

PELEA [7, 52] follows a continuous planning approach but, unlike other approaches [77, 20], it allows planning engineers to easily generate new applications by reusing and modifying the components as well as a high flexibility to compare different techniques for each module or even incorporate one's own techniques. It includes components to dynamically integrate planning, execution, monitoring, replanning and learning techniques. PELEA provides two main types of reasoning: high-level and low-level (mostly hardware). These two planning levels offer two main advantages: 1) both levels can be easily adapted to the requirements of the agent; and 2) replanning can be applied at each level, which grants a greater degree of flexibility when recovering from failed executions. It also enables the addition of more levels to allow developers for a more hierarchical decision process. We will consider only the high level reasoning of PELEA in this PhD Dissertation as it is sufficient to tackle most problems, as it has been shown in many robotics applications.

Figure 2.1 shows the main components of PELEA. PELEA is controlled by a module, called Top-level control, which coordinates the execution and interaction of the Execution and Monitoring components. In the following, the life-cycle of the architecture is described.

- *Execution module*. The Execution is the starting point of the architecture, and it is initialized with a planning task, which current state is read from the environment through the sensors. The environment is either a hardware device, a software application, a software simulator, or a user. The Execution is responsible of reading and communicating the current state to the rest of modules as well as executing the actions of the plan in the environment.

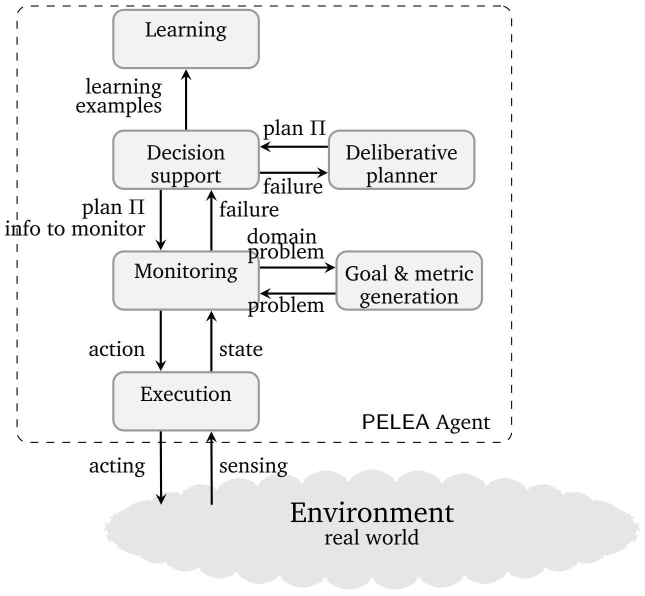

Figure 2.1: Architecture of a PELEA agent.

- *Monitoring module*. The main task of the Monitoring is to verify that the actions are executable in the current state before sending them to the Execution. The planning task is sent by the Execution to the Monitoring. The Monitoring calls the **Decision Support**, which in turn calls the **Deliberative Planner**, to obtain a plan and the *info to monitor*. The actions in the plan are directly sent to the executor. Once the actions are executed, the Monitoring receives the necessary knowledge (current state, problem and domain) from the Execution. When the Execution reports the Monitoring the state resulting from the execution of some action of the plan, the process called *plan monitoring* starts. This process verifies whether the values of the variables of the received state match the expected values or not. As a first step, the Monitoring checks whether the problem goals are already achieved in the received state. If so, the plan execution finishes; otherwise, the Monitoring checks whether the received state matches the expected state or not (with the info to monitor received by the Decision Support) and determines the existence of a plan failure or not.

The *info to monitor* parameter, provided by the Decision Support, comprises the information that needs to be monitored to guarantee a successful plan execution. Specifically, it includes: i) the variables to be monitored, i.e. those that are directly related to the plan, ii) the time at which the variable is generated, and the earliest and latest time at which the variable will be used, respectively; and iii) the value range for each variable, denoting the set of correct values that the variables can take on.

- *Goal & metric generation module*. The goal & metric generation module is designed to automatically select the new goals and metrics to be used according to the current state of the execution. This module is invoked by the Monitoring in case the system decides to change dynamically the goals or metrics along the plan execution. A common use of this module is for oversubscription problems [103], where not all goals can be satisfied. This problem is generally solved by choosing some goals and discarding others either online or offline.

- *Decision support module*. It selects the variables to be observed by the Monitoring during the plan monitoring (info to monitor), and takes a decision of repairing the current plan or re-planning from scratch when the Monitoring detects a plan failure. The decision between repairing or replanning is done through the application of an anytime plan-adaptation approach [45], which uses a regressed goal-state heuristic. A regressed goal state is a tuple of the form $G = \langle L, t \rangle$ where L is the set of atoms, i.e. values of the state variables, and t is the time of G, which usually coincides with the start time of one action (sequential planning) or more than one action (parallel planning). The heuristic estimates the best G according to parameters as the cost or stability of the estimated plan. Then a new problem from the current state S to the selected regressed goal state is generated and the planner is invoked. Note that the first G is the one from which the whole original plan can be reused;

the subsequent goal states represent reachable states from which to reuse ever decreasing parts of the original plan; and the final G entails no reuse of the plan at all.

The Decision Support is capable of deciding between replanning or repairing in a timely fashion. It uses an algorithm with anytime capabilities whereby a first solution plan is rapidly returned, and the solution quality may improve if the algorithm is allowed to run longer [45]. The heuristic takes a balanced response between metric (cost or makespan) and plan stability (part of the original plan that can be reused in the new solution plan) [40]. Plan stability is one of the principal reasons for claiming the preference of plan repair over the alternative of replanning. The heuristic estimates an approximate plan Π_{replan} (a plan from S to the last regressed partial state G_n discarding the whole original plan), and a plan Π_{repair} (a plan from S to the G_1 keeping the whole original plan). If cost(Π_{replan}) $<$ cost(Π_{repair}) or stability(Π_{replan}) $>$ stability(Π_{repair}) then replanning is chosen as the preferred option. Otherwise, the algorithm analyzes the cost and stability of the subsequent regressed goal states (G_2,..., G_{n-1}) and maintains the best regressed goal state computed so far until time expires. Once a goal state G_i is selected, the Deliberative Planner is invoked with the initial state S and goal state G_i. The returned plan is concatenated with the plan tail of the original plan taking into account the causal links and time constraints.

On the other hand, the Decision Support Module computes the info to monitor through an extension of the goal regression method proposed in [44], which is inspired by the mechanism used in triangle table defined in [36]. This mechanism is only used so far to monitor the correct plan execution. The Decision Support also communicates the Monitoring with the Deliberative planner module and retrieves training instances from the execution and the plans to be sent to the Learning module.

- *Deliberative planner module*. It receives a planning task and generates a plan. This module is invoked when the Decision Support has to fix (repair/replan) a plan. In this latter case, the state of the planning task will be the current observed state. Several planners have been successfully used for this module: LPG-TD [48], CRIKEY [23], TFD [33] and LAMA [90].

- *Learning module*. It infers knowledge from a training set sent by the Decision Support module. The knowledge can be used either to modify the domain planning model or to improve the planning process (heuristics). Apart from the different levels of reasoning, PELEA may also learn from past executions and reason about the current problem to improve its efficiency.

The components run as separate processes and communicate through sockets. The inputs are defined by the PDDL domain language. The knowledge exchanged among components follows the domain-independence principle with domain-independent APIs (through XML). Here lies the generality of PELEA; one can exchange a component and PELEA will continue working as it is, maintaining the XML APIs and their semantics, which are the standard ones in planning: actions, goals, states and plans.

2.1.2 Multi-agent frameworks

In this section, we highlight the most relevant planning and execution architectures that were adapted or applied to a multi-agent context. The frameworks that we present here are mostly related to general problem solving and are aimed at coordinating activities of several agents in dynamic environments, with no particular focus on planning and execution. However, we believe it is worth revising some of the most outstanding architecture prototypes that somehow combine planning and execution functionalities.

An agent is a physical or virtual entity that can act, perceive its environment (in a partial way) and communicate with others, it is autonomous and has skills to achieve its goals. An agent is regarded as an entity that is constantly executing in an context, interacting with other agents, and it is able to react to changes in the environment by reasoning. A *Multi-Agent System* (MAS) comprises an environment, objects and entities or agents (the agents being the only ones to act), relations between all the entities, a set of operations that can be performed by the entities and the changes of the universe in time and due to these actions [34]. A MAS is thus defined as the environment shared by several agents that interact for solving either a private or public task.

A MAS approach is adopted when the task is too complex to be solved as a monolithic application or because none of the agents have a complete view of the problem. This typically occurs when there exist specialized agents and/or the problem information is distributed. In either case, a development tool for MAS and a common language for agents to interact and communicate with each other are required when building a MAS.

In this section, we summarize the most relevant languages and platforms for the construction of a MAS. A former survey on the programming languages and tools for MAS can be found in [14].

2.1.2.1 Agent Communication Languages

Agents communicate with each other in order to exchange information. When communicating with other agents, an agent uses a specific Agent Communication Language (**ACL**). Much work has been done in developing ACLs that are declarative, syntactically simple and readable by people.

The Knowledge Query and Manipulation Language (**KQML**) is one proposed standard of ACL for exchanging information and knowledge among software agents. This proposal is part of the ARPA Knowledge Sharing Effort aimed at developing techniques and methodologies for building large-scale knowledge bases which are sharable and reusable [37]. KQML is both a message format and a message-handling protocol to support run-time knowledge sharing among agents. KQML provides an extensible set of *performatives*, which defines the permissible operations (*speech acts*) that agents perform on each other's knowledge and goal stores. Higher-level models of interagent interaction such as contract nets and negotiation are built using the *performatives*. Additionally, KQML provides a basic architecture through a special class of agent called *communication facilitator* which coordinates the interactions of other agents to support knowledge sharing.

The Coordination Language (**CooL**) [11] is another ACL proposal which relies on speech act theory-based communication for describing a coordination protocol based on a multi-agent plan. **CooL** integrates the agents dialog in a structured conversation framework that captures the coordination mechanisms that agents are using when working together. Other languages like **AgentTalk** introduce an inheritance mechanism into the protocol description to incrementally define new protocols based on existing ones and implement customized protocols to suit various application domains [63].

The aforementioned ACLs were later superseded by the FIPA-ACL proposal [85, 1], supported by the Foundation for Intelligent Physical Agents (**FIPA**). FIPA-ACL is likely to be the most widely used ACL and has become de facto a standard ACL in MAS.

In summary, ACLs provide the necessary functionalities for defining communication protocols in MAS and for agents to exchange information, thus enhancing relevant aspects such as agent collaboration. The agent-to-agent language communication is key to realizing the potential of the agent paradigm, just as the human language was key to the development of human intelligence and societies.

2.1.2.2 Platforms for building Multi-Agent Systems

Here we present some of the most recent and significant multi-agent platforms for building a MAS, highlighting the communication capacities of the platforms.

AgentScape [3] is a middleware layer that supports large-scale agent systems. It is a natural extension of logic programming for the BDI [98] agent architecture, and provides an elegant abstract framework for programming BDI agents. The rationale behind the design decisions are (i) to provide a platform for large-scale agent systems, (ii) support multiple code bases and operating systems, and (iii) interoperability with other agent platforms. The AgentScape model also defines services which provide information or activities on behalf of agents or the AgentScape middleware. AgentScape is adaptive and reconfigurable and it can be tailored to a specific application (class) or operating system/hardware platform.

JADE (Java Agent Development Environment) [5] is a framework to easily develop intelligent multi-agent system applications in agreement with the FIPA specifications. JADE is an Open Source project which purpose is to simplify development of agents while ensuring standard compliance through a comprehensive set of system services and agents. The JADE system can be split into several hosts where each host executes only one Java Virtual Machine and implements an agent as one Java thread. Parallel tasks can be executed by one agent and are scheduled by JADE in a cooperative way. We can describe the JADE system from two different points of view: (i) JADE helps application agents to exploit some feature covered by the FIPA standard specification, like message passing or agent life cycle management and (ii) JADE is a Java framework that helps programmers to develop agent applications with the FIPA standard through object oriented abstractions.

EVE is a web-based agent platform [4] that is open and dynamic, i.e. agents can live and act from anywhere – in the cloud, on smartphones, robots and others. The agents communicate with each other using existing protocols (JSON-RPC) over existing transport layers (HTTP, XMPP), offering a simple platform solution. EVE defines an agent as a software entity that represents existing, external, sometimes abstract, entities such as human beings, physical objects, abstract goals. The agent runs independently of the entity it represents. This autonomous behavior requires a set of capabilities which are: (i) time independence, (ii) the possibility to keep in memory a model of the state of the world and (iii) a common language to communicate between agents. The time independence capability provides requests at scheduled times, thus instantiating the agent at any given time. The reason for giving a separate memory capability to the agents is that, in most implementations, EVE agents have a request based lifecycle. The agent is instantiated to handle a single incoming request and is destroyed again as soon as the request has been handled. Only the externally stored state is kept between requests. The agent identity is formed by the address of its state, not by an in-memory, running instance of some class.

JASON [6] is an interpreter for an extended version of AgentSpeak [87]. It implements the operational semantics of that language, and provides a platform for the development of multi-agent systems, with many user-customisable features. JASON is available Open Source, and is distributed under GNU-LGPL. The performatives that are currently available for agent communication in JASON are largely inspired by KQML. Some of the features available in JASON are (i) communication between agents based on speech-act, (ii) plan annotations that can be used to elaborate decision selection functions, (iii) fully Java customisable selection functions, trust functions, and overall agent architecture (perception, belief-revision, inter-agent communication, and acting), (iv) clear notion of multi-agent environments (this can be a simulation of a real environment) and other features.

MAGENTIX2 [107] is an agent platform for open Multiagent Systems which main objective is to develop technologies to cope with the (high) dynamism of the system topology and with flexible interactions, which are both natural consequences of the distributed and autonomous nature of the components. In this sense, the platform supports flexible interaction protocols and conversations among agent organizations. It provides support at three levels: (i) Organization level, technologies and techniques related to agent societies; (ii) Interaction level, technologies and techniques related to communications between agents; and (iii) Agent level, technologies and techniques related to individual agents (such as reasoning and learning). In order to offer these support levels, MAGENTIX2 is formed by different building blocks, and provides technologies to the development and execution of MAS. Magentix2 uses the Apache Qpid implementation of AMQP as a foundation for agent communication. This industry-grade open standard is designed to support reliable, high-performance messaging over the Internet. It facilitates the inter-operability between heterogeneous entities. MAGENTIX2 allows heterogeneous agents to interact to each other via FIPA-ACL messages, which are exchanged over the AMQP standard. Thus, MAGENTIX2 agents use Qpid client APIs to connect to the Qpid broker and to communicate with other agents at any Internet location. MAGENTIX2 incorporates conversational agents, which is a class of agents called CAgents. CAgents allow the automatic creation of simultaneous conversations based on interaction protocols. It also provides native support for executing JASON agents. Moreover, JASON agents can benefit from the reliable communication and tracing facilities provided by MAGENTIX2.

2.1.2.3 Architectures for Multi-Agent Systems

Our interest lies in systems that integrate planning and execution in a multi-agent context and, specifically, on distributed cooperative systems [97]. Hence, in this section, we present the most relevant multi-agent architectures related not only to planning and execution but to agent problem-solving in general.

Generalized Partial Global Planning [29] and its hierarchical task network representation TÆMS [28, 65] (**GPGP/TÆMS**) was developed as a domain-independent framework for on-line coordination of the real-time activities of small teams that cooperate to achieve a set of high-level goals. GPGP/TÆMS is primarily concerned with scheduling activities (coordination) rather than the dynamic specification and planning of evolving activities or the accomplishment of a common goal. In GPGP/TÆMS, each agent has a partial global plan and their local view of the activities they intend to pursue. In order to build and refine each partial global plan, agents communicate their local plans to the rest of agents. Coordination is achieved in terms of coordinating a distributed search of a dynamically evolving goal tree.

The focus of GPGP is on the distributed resolution of temporal and resource interaction between distributed tasks. TÆMS is an abstract model of the agent problem-solving activities that allows to abstract the internal representation of the domain problem solver into a TÆMS task structure for use by GPGP. GPGP/TÆMS deals with high-level coordination issues (scheduling and task selection) involving decisions for each agent about what subtasks of which high-level goals it will execute, the level of resources it will expend in their execution, and when they will be executed. GPGP/TÆMS is thus most appropriate for ***soft real-time applications*** operating in dynamic and resource-bounded environments where there are complex interdependencies among activities occurring at different agents.

The type of high-level coordination of GPGP/TÆMS is very different from the type of low-level, local and fine-grained coordination that commonly occurs when a protocol is created among agents to synchronize their activities. In frameworks such as CooL [11] and AgentTalk [63] the emphasis is on the flow of messages and how the dialog between agents is structured. Such frameworks generally combine finite state machines with enhancements, e.g. exceptions, to support a flexible and explicit specification of a communication process. In contrast, in GPGP/TÆMS the focus is on a domain-independent and quantitative evaluation of the interactions among tasks and the dynamic formation of temporal constraints to resolve and to exploit these interactions.

GPGP coordination is about exchanging the necessary information for a group of agents to analyze their activities so as to reformulate their local problem solving based on an understanding of the relevant activities of other agents; the purpose of this reformulation is to produce with fewer resources higher-quality solutions from the perspective of the group's problem-solving goals. While GPGPcontains protocols that define the exchange of messages and the flow of the conversation, the emphasis is not on the machinery for specifying the protocols but on the content and timeliness of the information that must be exchanged in order to create group performance that approximates what would be obtained if control of agent activities was centralized and done in an optimal manner. Particularly, it was a pioneer in the investigation of task/goal scheduling, resource allocation and timing constraint in MAS.

The REsuable Task Structure-based Intelligent Network Agents (**RETSINA**) [108] is a multi-agent infrastructure which consists of an open society of three different reusable agent types (*interface agents*, *task agents* and *resource agents*) that can be adapted to address a variety of domain-specific problems. The agents self-organize and cooperate in response to task requirements. The interface agents interact with the user, receive user input and display results; the task agents help users perform tasks by formulating problem-solving plans and carrying out these plans through querying and exchanging information with other software agents; and the resource agents provide intelligent access to a heterogeneous collection of information sources. Each agent draws upon a complex reasoning architecture that provides three crucial characteristics of the overall framework: (i) a multi-agent system where agents asynchronously collaborate with each other and their users,

(ii) agents actively seek out information and (iii) the information gathering is seamlessly integrated with problem solving and decision support. The RETSINA framework is implemented in Java and is being used to develop distributed collections of intelligent software agents that cooperate asynchronously to perform goal-directed information retrieval and information integration in support of a variety of decision-making tasks.

Intelligent Distributed Execution Architecture (**IDEA**) [76] is a multi-agent platform used by some NASA's space vehicles that provides agents with a unified representational and computational framework for planning and execution. The main components of IDEA are:

- **Tokens and procedures:** a token is the fundamental unit of execution. It is a time interval during which the agent executes a procedure, which can be executed when it receives all the input arguments. The time when the procedure starts is the token start time. At any time during the execution, the procedure can return the results to the output arguments. The execution of a procedure is terminated when a status value is returned or the agent interrupts the token's execution.
- **Virtual machine:** contains four main components which provide the basis for the agents autonomous behavior: the domain model, the plan database, the plan runner, and the reactive planner. The domain model describes which procedures can be exchanged with other agents, which procedure arguments are expected to be determined before a goal is sent to another agent (input arguments) and on which arguments the agent is expecting execution feedback from some other agent executing the token (output and status arguments). The Plan Database describes portions of old plans, the tokens currently in execution, and the currently known future tokens, including all the possible

ways in which they can execute. The Plan Runner is the core execution component of the agent. It is activated asynchronously when either a message has been received from another agent or an internal timer has gone off. The Reactive Planner is in charge of returning a plan that is locally executable. It must guarantee the consistency of token parameters in terms of the plan's constraints and support for the token according to the domain model.

- **Communication wrapper:** defines a simple communication protocol between separate IDEA agents; an underlying inter-process communication mechanism. It is in charge of sending messages that initiate the execution of procedures or receive goals that are treated by the agent as tokens.

Coupled Layered Architecture for Robotic Autonomy (**CLARAty**) [79] is a framework for generic and reusable robotic components that can be adapted to different robots. It has two layers, the functional layer and the decision layer. The first layer defines the abstractions of the system and adapts the abstract components to real or simulated devices, providing the algorithms with low- and mid-level autonomy. It uses object-oriented system decomposition and employs a number of known design patterns to achieve reusable and extendible components. The Decision Layer is in charge of reasoning about global resources and mission constraints, proving to the systems with high-level autonomy. It reasons globally about the intended goals, system resources, and the state of the system and its environment. This layer plans, schedules and executes activity plans. It also monitors the execution, modifying the sequence of activities dynamically when necessary. The two layers interacts using a client-server model. The decision layer requests the functional layer about the availability of system resources in order to predict the resource usage of a given operation.

Planning and Execution Framework (PlanInteraction) [1] is a multi-agent planning and execution architecture where agents autonomously perform science targets, execute a set of tasks in a simulated or real-world, monitor the plan execution attending to potential discrepancies, and take decisions for repairing or re-planning in case of a plan failure. PlanInteraction is implemented by integrating PELEA [52] agents in an open MAP platform called MAGENTIX2 [106]. An agent in PlanInteraction can be defined with all the same functionalities of a PELEA agent or only some of them. Particularly, in this work, we will distinguish between two types of agents, *planning agents* and *execution agents* (see Figure 2.2). A planning agent is in charge of calculating plans for solving a planning task. An execution agent comprises all the machinery necessary for monitoring the execution of a plan and providing a reactive response to a plan failure (this is a new module that we introduce in Chapter 3). Figure 2.2 shows the particular configuration of PlanInteraction that we use in this Ph.D. dissertation. As we can see, the architecture consists of three main modules:

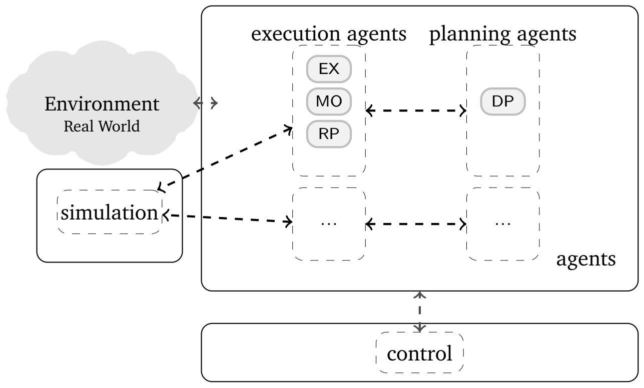

Figure 2.2: Multi-Agent PlanInteraction Architecture. EX: execution; MO: monitoring; RP: reactive planner; DP: deliberative planner.

- *Control:* is responsible for registering agents in the system, initializing the internal clock, and controlling the conditions for the system termination. The

[1] *http://planinteraction.cguz.org/*

internal clock manages the time of all agents in the system.

- *Simulation:* represents the simulated state of the world. It works in the system as a *simulator agent* that provides agents access to the information in the simulated environment and modifies it through the execution of the actions in their plans.

- *Agents:* comprises the set of agents of the problem. An agent in PlanInteraction represents any combination of the PELEA modules. The composition of modules in each agent depends on the problem specification and agent's capabilities.

  Hence, the three modules *Execution* (EX), *Monitoring* (MO), and *Reactive Planner* (RP) are embedded in a single PELEA-like execution agent with capabilities for tracking plan execution and plan monitoring. We can also opt for creating planning agents that only comprise the *Deliberative Planner* (DP) module, thus providing agents with capabilities for planning and repairing plans (this is the configuration shown in Figure. 2.2).

  In some applications, we might even want to have several different Execution modules but a single Monitoring; for example, a robotics application where one monitor controls the operations of several robots moving in a shared space.

In PlanInteraction, we distinguish three different coordination levels:

1. *Planning-Planning Coordination* between planning agents when they have to generate jointly a solution plan for a particular task. This happens when the planning task is defined by multiple entities (agents) which are distributed functionally or spatially, and capabilities are distributed across the agents domains. The emphasis of this coordination is on the design of a consistent, joint solution plan among several planning agents before execution.

2. *Execution-Execution Coordination* focuses on the coordination between execution agents when they attempt to solve a plan failure at execution time. Typically, a plan failure occurs when planning coordination has not been applied and execution agents attempt to execute their plans in a common environment, when the environment changes unexpectedly, or when the execution of an action does not report the expected results. When this happens, agents attempt to communicate between them to handle the plan failure.

3. *Execution-Planning Coordination* takes place when execution agents are not capable of reaching a solution during the execution-execution coordination, and so they have to resort to their planning agents to find a new plan that solves the task.

In this PhD dissertation, we will work on coordination levels 2 [55, 53] and 3. The coordination level 1 is not necessary because each planning agent computes its individual plan without coordinating with others. More specifically, the emphasis of this work is on the coordination level 2.

2.2 Approaches to planning and execution

Classical planning techniques are applicable in deterministic and static environments. Planning in a deterministic environment assumes that the execution of an action always leads the system to a single other state that reflects the correct execution of the action. In addition, if the environment is static, the planner is not concerned with any exogenous event; that is, the only events or modifications that can happen in the world are due to the effects of the actions.

Non-deterministic and dynamic environments, in contrast, assume the occurrence of exogenous events and the possibly wrong execution of the actions. In non-deterministic environments where actions can have uncertain outcomes, it may be useful to model in the plan the fact that a machine component may fail and to plan the recovery mechanism to address the failure. This usually requires to model the domain as an stochastic system defining the probability distribution on each state transition (probability for each state and action over next states). Markov Decision Processes [70] (MDPs) are a widely popular approach to address planning in non-deterministic environments. MDPs can deal with settings where action outcomes are uncertain, they are used to model sequential decision-making scenarios with probabilistic dynamics and they can even be extended to deal with partially observable worlds. MDPs calculate policies that specify the optimal actions for each agent for any possible belief state or partially observable state. In general, formal models for planning under uncertainty are inspired on the use of MDPs and they differentiate in the implicit or explicit communication actions of the agents and in the representation of agent beliefs [100].

However, in some scenarios accurate probabilities of the actions outcomes are not available or the abnormal cases of behaviour can be dealt with at the controller level. When it is not possible to have a model of the uncertainty in the world, MDPs are not applicable. Likewise, a Finite State Machine (FSM) can also be used to model a plan that may result in many different execution paths. A FSM usually requires an explicit modeling of a finite set of plans (states and transitions), although some approaches address this limitation with a method for deriving finite-state controllers automatically from models [13], and it is particularly aimed at choosing the next immediate action.

Such as we mentioned in our architecture PlanInteraction of Section 2.1.2.3, the MAP scenarios we focus on this work picture a set of independent execution and planning agents. Each execution agent is associated to a planning agent (deliberative planner). Hence, execution agents receive the plans that solve their tasks from the planning agents, and execute the plans in an unpredictable and shared environment. Unexpected events and faulty actions may prevent an execution agent from executing its plan. In such a case, the execution agent resorts to its reactive plan repair so as to resume the plan execution as soon as possible (single-agent plan repair) or otherwise the agent can resort to a collective repair process (multi-agent plan repair). The principal characteristics of our planning scenario are:

- the plans of the execution agents are computed by their associated planning agents; execution agents seek to execute their plans and achieve the goals of their tasks
- failures that happen during the plan execution are attended at the controller level; agents aim to quickly fix the plan failure and resume the execution
- unexpected events and faulty actions may happen frequently; execution agents adopt a repair mechanism that allows for short-term reactivity but also preserving the achievement of the goals

With these ingredients in mind, we focus on approaches of planning and execution and, more particularly, in plan repair for short-term reactivity. Whilst there is a large body of research on automated planning for both single-agent and multi-agent contexts, most of the techniques put the emphasis in the construction of a single (joint) plan to solve a task (using centralized approaches). We, however, are interested in techniques for calculating promptly recovery plans that adapt the plan under execution to the new context after a failure.

In this section, we revise the principal literature related to the aforementioned topics and present the main approaches to planning and execution. We particularly distinguish the techniques that are typically applied within the planning system (Section 2.2.1) and the ones that are intended to be used at the executive control (Section 2.2.2). Hence, the first group of techniques have a more deliberative flavour whereas the second group provide a more reactive nature.

All of the approaches presented in this section are particularly oriented to single-agent scenarios. In Section 2.2.3, we will extend this analysis to multi-agent contexts, including a brief discussion on the advantages and limitations of adapting the single-agent approaches to a multi-agent context.

2.2.1 Deliberative plan repair and replanning

This section details the deliberative classical planning techniques that have mostly influenced this Ph.D. thesis.

When a plan cannot execute due to a variation in the environment or an unexpected event, two general mechanisms exist for adapting the plan to the new circumstances, plan repair or replanning. While the former puts the focus on repairing only the affected part of the plan while retaining as much as possible of the unchanged parts of the plan, replanning triggers a new planning problem with the current situation of the environment as the new initial state and generates a new plan from scratch.

Many algorithms exist for repairing or adapting the current plan [40, 35, 45]. Sometimes plan repair and replanning are commonly used as a basic planning and execution strategy to deal with dynamic domains, working as follows: the agent receives a plan that achieves the goal under some constraints based on the available (often incomplete) information. Then, the agent executes the actions of the plan as long as the plan remains valid. Otherwise, the agent interrupts the execution and attempts to repair the plan or resorts to replanning. Once the plan is fixed or a new plan is computed, it is sent to the agent and the execution keeps on until goal completion. The main drawback is that this solution is unaffordable for systems that require to react quickly in unpredictable environments. In [69], they employ this basic planning and execution strategy. Specifically, they have an stationary robot that waits for a plan. A global planner uses an A* algorithm that plans directly in a two-dimensional cost map and does not consider the movements of the robot [62]. The fact that it does not consider the dynamics of the robot guarantees that the global planner returns a plan quickly. Nevertheless, the planner is optimistic in the plans that it creates.

Other planning approaches deal with dynamic domains, but they do not offer quick responses at runtime execution. Contingent planning [84] generates plans where some branches are conditionally executed depending on the information obtained during the execution. Another approach is conformant planning [58], which allows to deal with uncertainty on the initial conditions and the action effects without monitoring the plan execution. However, these approaches cannot consider all possible contingencies and the computation time is often prohibitive. In order to avoid the computational effort of considering all possible unexpected situations during planning time, the continual online planning approaches tackle these situations only when they appear; thus, when pre-computed behaviors are not available, an additional recovery mechanism has to react quickly to unexpected events.

Gerevini presented in the context of the plan repair the planner LPG-Adapt [49, 50]. The aim of LPG-Adapt is to modify the original plan by adapting it within limited temporal windows. Each window is associated with a particular subproblem that contains some *heuristic goals* facilitating the plan adaptation. A small temporal window refers to plan repair, and when the length of the temporal window becomes the length of the whole plan of actions it degenerates into a replanning strategy. In this latter case, the approach performs worse than replanning from scratch due to the overhead for building smaller replanning windows (subproblems).

LPG-Adapt represents the current plan through a Graphplan structure [12]. Hence, instead of searching for a new solution when a failure occurs (as it is usually done in classical planning), the structure is exploited for reasoning on (and reusing) the current plan. One of the weaknesses of the approach lies in the construction of the Graphplan because it requires a (potentially expensive) phase for grounding the actions at hand, which causes a non-negligible blow-up of the Graphplan structure. However, the proposed solution represents an efficient approach for a plan computation.

In the context of replanning, any deliberative planner could be used. Some valuable ideas from deliberative planning can be gathered and adapted to reactive systems. This is the case of the planner heuristic-based Fast Downward (FD) [57] planner, which uses a *multi-valued* representation for the planning tasks instead of the more common propositional representation. FD makes use of SAS+ like [9] state variables to model the facts that conform states. For each state variable, FD infers its associated Domain Transition Graph (DTG), a structure that reflects the evolution of the value of a variable according to the actions of the task. The information of the DTGs is compiled into the Causal Graph, which displays the dependencies between the different state variables. FD applies a best-first multi-heuristic search which alternates in an orthogonal way the heuristics of FF [59] and its CG [57], a heuristic estimator calculated utilizing the Causal Graph. A SAS+ representation is a compact a more efficient way of representing states and as such it can be helpful in reactive systems.

2.2.2 Reactive planning and execution

Reactive planning and execution is related with dynamic and unpredictable environments where failures occur frequently and the *execution* agent needs to react quickly. In the following of this section, we focus on reviewing some techniques that are intended to be used at the executive control.

2.2.2.1 SimPlanner

Sapena and Onaindia [82, 95] developed an integrated tool called SimPlanner for planning and execution monitoring in a dynamic environment. SimPlanner allows interleaving planning and execution. The user monitors the execution of a plan, interrupts the monitoring process to introduce new information from the world and activates the plan repair process to get it adapted to the new situation. The objective of SimPlanner is to avoid generating a complete plan each time a failure occurs by retaining as much of the original plan as possible.

The plan repair technique used in SimPlanner is based on reachable states and the construction of a relaxed graph [59] that builds an approximate plan. Specifically, SimPlanner uses an heuristic function based on a relaxed graph to find an optimal reachable state that is used as a subgoal and then builds a plan from the new situation to the reachable state. Finally, this plan is concatenated with the rest of the old plan from the optimal reachable state. The repairing module of SimPlanner returns a recovery plan that is always optimal or close to the optimal solution.

An extension of SimPlanner was later proposed in [96], where given a state and a deadline, the planner searches for an action executable in that state, which can successfully lead to a goal state. The process is repeated, starting from the resulting state, until the execution of an action reaches the goal state. The scheme is flexible and can be carried out in an interleaved way along with the execution turning it into a mix of reactive (it can execute the action during the planning process) and deliberative (it employs heuristic functions to obtain a complete plan) planner. As a drawback, SimPlanner does not offer any guarantee that the first initial solution will be executable, which is a key factor in reactive planning.

2.2.2.2 Task Control Architecture

The Task Control Architecture (TCA) [101] was introduced in Section 2.1.1.1. Here we focus on the execution control of the architecture.

TCA uses an exception handling mechanism to detect and repair plan failures. When an exception is triggered, TCA searches up in a task tree, starting from the node where the exception happened, to find a handler for that exception. The task tree is not created dynamically and is often domain and context-dependent; i.e., the same exception may need to be handled differently, depending on the environment and where the exception occurs. The idea of a pre-calculated search space as a task tree is interesting and also efficient in systems that require a quick fix at runtime when a plan failure arises. Creating the search space dynamically and irrespective of the domain context is also attractive for planning and execution systems but, obviously, the process to generate the search space should be time-bounded.

2.2.2.3 Intelligent Distributed Execution Architecture (IDEA)

The multi-agent framework IDEA [38], explained in Section 2.1.2.3, implements the executive control of an agent with a reactive planner and all agents use the same reactive planner as their base. The reactive planner is a search process guided by a simple heuristic-based chronological backtracking engine. As in [49, 50], IDEA controls the speed of the reactive planner throughout the length of the horizon (temporal windows in Gerevini) over which the reactive planner is requested to repair a plan. When the horizon becomes shorter, the size of the reactive planning problem becomes smaller and so the size of the planning search space to generate. In other words, the smaller the planning horizon is, the more reactive the executive control of the agent of IDEA becomes. The horizon reductions in IDEA do not offer any guarantee to achieve a correct plan execution; that is, building a plan with a small plan horizon (e.g., one action) can affect future horizons and the achievement of future goals. The reason is that the reactive planner of IDEA has only visibility on subgoals and tokens that occur during one planning horizon. Nevertheless, focusing only on one portion of the plan is an appropriate way of addressing failures in reactive planning and execution systems.

2.2.3 Multi-agent planning and execution

In general, there are not many approaches in the literature that address the topic of multi-agent planning and execution. Most of MAP systems tackle the issue of agent interaction at a planning level, or deal with execution agents but reactive techniques are implemented at a single-agent level.

Recently, there has been a growing interest in the design of MAP systems [27], also boosted by the celebration of the first competition of Distributed and Multiagent Planners (CoDMAP[2]). CoDMAP is not concerned with agent execution but only with the design of techniques and algorithms for multi-agent planning. In CoDMAP, planners participated in one of the two tracks of the competition, the centralized track or the distributed track. Most planners that participated in the distributed CoDMAP track implemented a multi-threaded or distributed search by using TCP/IP protocols for agents communication [105, 109]. The only planner that used of a middleware multi-agent platform for providing agents the required communication services is *Forward Multi-Agent Planning* (FMAP) [110]. Particularly, FMAP builds upon Magentix2 [107], a multi-agent platform which was explained when the PlanInteraction architecture was introduced in Section 2.1.2.3.

In the following, we will put the focus on MAP systems that involve both planning and execution, specifically highlighting the distribution of the planning information across agents, the development of efficient distributed planning algorithms and the aspects related to agent communication either during planning or execution.

2.2.3.1 Micalizio and Torasso

Micalizio and Torasso [73] present an approach to team cooperation for plan recovery in multi-agent systems. At planning time (before execution), agents are organized in teams that cannot change during the plan execution. A completely instantiated multi-agent plan is calculated by a human by exploiting a central planning tool. Then, a dispatcher module splits the information of the multi-agent plan in as many sub-plans as available agents. Each team of agents is responsible of executing some actions of the multi-agent plan to achieve a specific portion of the global goal. Hence, at each time instant multiple actions are executed concurrently, and each agent monitors the progress of the sub-plan it is responsible for.

[2]http://agents.fel.cvut.cz/codmap/

In case the monitor detects a failure in an action, the agent repairs (if possible) its own subplan by means of a deliberative replanning mechanism. When this mechanism is unable to find a recovery plan for the subplan, a team level recovery strategy is invoked. The team strategy is based on the cooperation of agents within the same team. Specifically, the agent in trouble (requesting agent) asks only one agent (cooperating agent) to cooperate for recovering from a particular action failure. The cooperating agent is properly selected by means of the *services* information. In this approach, the service information is one of the effects of an action which results to be a precondition for other action(s). The services information are manually shared during the planning time. Another feature of this approach is that all the agents shared the same knowledge of the environment.

This team recovery approach involves only two agents of the same team, who have previously shared their information. As a whole, the architecture and control flow of the approach in [73] works similarly to our PlanInteraction architecture. However, our interest lies more in the development of an autonomous distributed system where agents can generate their own initial plan rather than using a deliberative central planner. Also, in our approach agents can request collaboration to any other agent available in the system rather than restricting to only one agent of a fixed team.

2.2.3.2 Jonge's approach

Jonge, Roos and van den Herik [24, 25, 26] present a protocol for plan repair in multi-agent plan execution. Their approach relies upon two general types of plan diagnosis: one primary plan diagnosis for identifying the incorrect or failed execution of actions, and a secondary plan diagnosis for identifying the underlying causes of the faulty actions. The plan-execution health repair is achieved by adjusting the execution of the agents' tasks instead of replanning the tasks. In order to tackle this, the plan diagnosis mechanism provides the information necessary for adjusting the plan, determines the agents that must be repaired so as to avoid further failures and identifies the agents responsible for plan-execution failures.

In Jonge's approach, adjusting the tasks plan is associated to plan repair through replanning [93, 112] and to the TÆMS task descriptions [86] for handling uncertainty in plan execution. Moreover, the protocol is conceived as a part of a distributed continual planning system [86], interleaving planning and execution by multiple agents. Jonge's approach assumes that agents have knowledge not only about their plans but also about the other agents' plans and applies the repair technique applies repair within the margins of the current plans.

Specifically, the repair technique is formulated as a constraint satisfaction problem that follows a 3-stage multi-agent protocol:

1. **Local solving.** The agents attempt to solve the violated constraints locally with the condition that variables' values of agents that are not involved in a constraint violation are locked, avoiding new conflicts with local recovery solutions. Any change in variable domains' values is updated.

2. **Communicate new domain.** For each constraint in which the agent is involved and for which the related variables have an altered domain, each agent communicates the new domain to the other agents involved in the constraint. Subsequently, the agents adjust the local constraints representing the influences of the plan-constraints. They repeat the two steps until the domains do not change anymore.

3. **Deliberative multi-agent search.** Agents agree on the order in which the agents will search for a value assignment of a variable. The first agent in the order assigns a value to its variable and communicate it to the agents involved in the conflicting constraints. Subsequently, these agents adjust their knowledge and decide the value assignment is valid. If successful, the second agent in the order searches for a value assignment, and so on. However, if the value assignment is not accepted, the search process backtracks, and the

previous agent has to find a new assignment. Eventually, a solution for the constraint satisfaction problem may be found and subsequently a plan health repair can be established.

In summary, the planning and execution model of this approach introduces a novel n-agent process for plan recovery, which is more deliberative than reactive. Reactiveness comes down to inserting only a small number actions to repair the plan. However, no results are presented in the papers and so we can regard this approach more like a proof of concept.

2.2.3.3 Wooldrige and Nicholas's approach

Wooldridge and Nicholas [115] present a model of cooperative problem solving (CPS) where some agents recognize the potential for cooperation with respect to one of their goals through team action. In this approach, agents communicate the information about their plans at execution time.

The CPS model characterizes the mental states of the agents that lead them to solicit, and take part in, joint actions. Overall, the process follows the next four steps:

1. **Recognition.** The CPS process begins when some agent recognizes the potential for cooperative action. This recognition may come about because an agent has a goal that it cannot achieve, or the agent has a common goal that requires a cooperative solution.

2. **Team formation.** During this stage, the agent that recognized the potential for cooperative action at the previous step solicits assistance. If this stage is successful, then it will end with a group of agents having some nominal commitment to collective action. The authors do not present any formal planning model.

3. **Plan formation.** Agents agree to a joint plan, which they believe will achieve the desired goal. The notation to form the plan is based on a branching-time tree formed of states connected by arcs representing primitive actions. Paths through the tree are sequences of actions with some conditions, which may denote agents, groups of agents, sequences of actions, and other objects in the environment.

4. **Team action.** Finally, the agents execute the newly agreed plan of joint action, which maintain a close-knit relationship. This relationship is defined by a convention, which every agent follows. In other words, in the plan execution, each agent play out the roles it has negotiated.

The steps of the CPS process are similar to those used in implementation-oriented models, like the Contract Net protocol [104]. The main feature of Wooldrige and Nicholas's approach is that the agents choose at runtime to work together to achieve a common goal and they normally execute joint actions. Joint actions are actions that require two executor agents, for instance, we might find a group of agents working together to move a heavy object, build a house, or write a joint paper. Despite our focus is not on joint actions and voting mechanisms (we assume our agents are altruistic), the rationale behind the CPS process and the Contract Net Protocol may be helpful for our purposes, specifically the formation of a cooperative group of agents for achieving a failed goal of an agent.

As a final remark, this approach identifies a set of critical points where the cooperation can fail:

- **Before forming the team:** an agent that has recognized the potential for cooperation may be unable to form a team.
- **During the team formation:** the agents may be unable to agree upon a plan of action.

- **During team execution:** cooperation may fail after a plan has been agreed because of unforeseen circumstances or because one of the agents drops its commitment to the endeavor.

However, authors do not present any recovery solution (repair or replanning) when cooperation fails. In this Ph.D. dissertation, we advocate for the idea of offering recovery solutions for some of the points where the cooperation can fail.

2.2.3.4 GPGP/TAEMS

GPGP/TÆMS [28, 65] is a framework where agents have common knowledge and use the same technique to achieve common goals, so the utility produced by one agent has equal value to another agent. Agents in GPGP/TÆMS share the local view of their plans or activities and a partial global view of the general plan for all the agents. The partial global view may be incomplete. More specifically, the shared TÆMS representation is used by the problem-solving, coordination and scheduling components of the framework to communicate among themselves. The TÆMS representation is an enriched representation of a goal tree that includes quantitative information and temporal sequencing constraints plus more dynamic information on the state of subgoal scheduling execution. The local view can be augmented by static and dynamic information about the activities of other agents and the relationship between these activities and those of the local agent; thus, the view evolves from a local view to one that is more global.

In general, the idea of GPGP/TÆMS of having a local view and a global view is not affordable in reactive planning and execution because it requires more memory and time consumption of CPU. Representing the view of the plan with a tree structure can be very helpful. Normally, searching in a tree structure is faster than other graph structures like, for instance, a finite state machine structure.

Regarding the recovery process, if a plan failure occurs, the agent uses its deliberative planner to recover from the failure. Whenever a solution is not possible with the deliberative planner, the agent activates its multi-repair mechanism where the agent requests assistance from other agents. The other agents employ the local view and partial global views to find a solution by calling a deliberative planner. A negotiation between the agents is always required. The failing agent only communicates with those agents having goals that are directly related to its goals.

As a conclusion, Table 2.1 summarizes the main essential features of the approaches more relevant to multi-agent reactive planning and execution that studied in this Section.

Table 2.1: Summarize of the approaches more related to multi-agent reactive planning and execution.

Author	**Planning time** (before executing)		**Execution**			
	goals	**multi-agent planning**	**multi-agent repairing**	**approach**	**team formation**	**knowledge**
Micalizio [73, 74]	common	yes	yes	deliberative	yes, before executing	common
Jonge [24]	individual	yes	yes	deliberative	yes	semi
Wooldridge [115]	common	yes	yes	deliberative	no	common
GPGP/-TAEM [65]	common	yes	yes	deliberative	yes	common
our approach (MARPE)	individual	no	yes	reactive	yes	individual

In general, the reviewed approaches presented the following features:

- Most of the approaches have common goals because they use a central planner to generate the initial plan.
- A multi-agent planning technique generates the initial plans. The planning technique in Jonge's approach generates a plan for each agent considering the preconditions that are relevant for the other agents' plans. In the approaches of Micalizio [73, 74] and Wooldridge [115], the multi planning technique is implemented with a central planner agent that determines the best initial plan.

Our approach, however, accounts for independent agents which have their own autonomy but that, on the other hand, operate in the same environment.

- The multi-repair process in all the approaches is oriented to deliberative planning rather than reactive planning. Moreover, all the approaches employ deliberative techniques for single repair planning. Deliberative planning is not affordable in dynamic and unpredictable environments where failures occur frequently and the execution agent needs to react quickly (reactive planning). This is the type of environments that motivated the research of this Ph.D. dissertation.
- Most of the approaches use the concept of team during the plan execution.
- As for knowledge, in most approaches agents have the same common knowledge, and thereby, the same capabilities. In our approach, we stick to the concept of individual and independent agents that have different knowledge of the environment and different capabilities. Besides, our purpose is to design a reactive model that follows this type of agents.

With all this in mind, we define the type of problems that we want to solve in this Ph.D. dissertation. We are interested in solving problems where:

- Agents have different knowledge and different capabilities.
- Each agent generates its own initial plan, which may conflict with the plan of the other agents.
- Agents share their capabilities before plan execution.
- A plan failure may be provoked by exogenous events or conflicting plans.
- Each agent repairs its plan failure with a single-agent recovery process, whenever possible, that allows to resume its plan execution

- The cooperative multi-repair process is activated during the plan execution if the agent is not able to find a solution plan with its recovery process.
- Agents decide when, how and with whom to cooperate with at runtime.

2.3 Conclusions

In this chapter several different planning and execution control architectures have been analyzed. For this analysis, we classified the architectures into single-agent and multi-agents architectures. The revision aims to find the desirable features for improving existing architectures. We have also included in the analysis two architectures developed in our research group, PELEA, a single-agent planning and execution architecture, and PlanInteraction, a multi-agent planning and execution architecture as an extension of PELEA. The two developed architectures allow to easily add new modules, such as our *reactive planner* module.

Moreover, some approaches to repair plans have been analyzed. We reviewed single-agent techniques to recover from a plan failure which are typically applied within planning systems, and techniques that are intended to be used in the executive control. The first group of techniques have a more deliberative flavor whereas the second group provide a more reactive nature. We also extended this analysis to multi-agent environments.

From the analysis of the multi-agent techniques, some important features have been identified. Most of the techniques rely on a deliberative multi-recovery process and so they are not able to deal with multi-agent reactive execution in planning applications. Typically, these techniques are applied in contexts in which the knowledge of the problem is common to all the agents and agents feature certain capabilities to detect and repair failures. The conclusions derived from the analysis helps us motivate and pose the requirements of a new multi-agent reactive planning and execution model that incorporates unexplored features such as a multi-agent reactive recovery process that is activated during plan execution and allow agents to recover from plan failures without affecting the plans of other executors of the environment.

In the following chapters, a general reactive planning and execution model that keeps track of the plan execution of an agent and handles an automated failure recovery is described. We also describe the multi-agent extension of this model that enables the continuous and uninterruptedly flow of the agents when a failure arises.

Chapter 3

Reactive Planning and Execution Model

"The vast majority of people are unthinking prejudice machines."

(Stefan Molyneux)

The focus of this chapter is on the development of a reactive planning and execution system capable of providing fast deliberative responses for repairing a failed action. Unlike most reactive systems, our system does not simply return the immediate next executable action [38], but it operates over a portion of the plan or *planning horizon*. The system also computes at runtime a search space that permits the agent to continuously repair failures during execution on any action within the planning horizon.

In this chapter, we propose a general *Reactive Planning and Execution* (RPE) model that keeps track of the plan execution of an agent and handles an automated failure recovery. The RPE system incorporates a reactive planning module that is exclusively used for plan repair, and it is independent of the deliberative planner which is used to compute the agent's initial plan. Unlike the integrated planning and execution approaches mentioned in Chapters 1 and 2, we propose a highly modular and reconfigurable architecture.

This chapter is organized as follows: Section 3.1 describes the architecture in which the RPE model is embedded. Next, we describe a planetary Mars scenario which will be used as an illustrative and running example throughout this chapter. Section 3.3 introduces the planning concepts that are necessary to define the RPE model, which is explained in detail in Section 3.4. Section 3.4.3.1 explains the process to estimate the time of the time-bounded deliberative process used by the reactive planner. Finally, Section 3.5 concludes and outline some discussions.

3.1 Architecture of Planning and Execution

In this section, we present the single-agent architecture which our RPE model relies upon. Single-agent architectures are aimed at solving problems which feature a single entity with reasoning and acting capabilities. Specifically, we build on the single-agent PELEA architecture detailed in Section 2.1.1.6 and we introduce some modifications and extensions over the PELEA architecture [52].

One first modification is that we split a PELEA agent into a *planning agent* and an *execution agent* as a functional classification, separating the planning capabilities of the agent from its executive machinery. The planning agent integrates the deliberative components of PELEA, i.e. the Decision Support module and the Deliberative Planner; and the execution agent features the Execution and Monitoring modules. In our RPE model, we integrate a **Reactive Planner** module that endows the execution agent with fast responsiveness to recover from plan failures. Hence, the components of an execution agent in our architecture are (see Figure 3.1):

- *Execution module*. This module offers the same functionalities of the Execution module defined in a PELEA agent; that is, reading the current world state from the environment through the sensors, communicating the current state to the rest of modules as well as executing the actions of the plan.

- *Monitoring module*. This module also keeps the same functionalities of the Monitoring module defined in PELEA, except that the parameters to monitor (*info to monitor*) are calculated now by the Monitoring instead of the Decision Support module. Moreover, when the Monitoring determines the existence of a plan failure, it informs first the Reactive Planner.
- *Reactive Planner module*. The Reactive Planner is the new component of our architecture. It is a planner capable of calculating a fast and reactive response when it receives notification from the Monitoring about a plan failure. The Reactive Planner uses a pre-computed search space, called *repairing structure*, to promptly find a plan that brings the current state to one from which the plan execution can be resumed. This will be explained in detail in Section 3.4.

The Execution, Monitoring and Reactive Planner of an execution agent operate the architecture of the RPE model. The control flow of the architecture is shown in Figure 3.1. An action plan Π for solving a planning task is calculated by the planning agent. Π consists of a series of actions to be executed at given time steps, each of which makes a deterministic change to the current world state. For instance, let's assume the execution agent receives the plan Π of Figure 3.2 composed of five actions $[a_1, a_2, a_3, a_4, a_5]$. The agent executes the action a_1 at time step t_0, a_2 at time step t_1, a_3 at t_2, and so on. The elapsed time from one time step to the next one defines an execution cycle; i.e., the monitor/repair/execution cycle of an action execution [46]. An execution cycle is usually interpreted as the minimum latency interval starting at the current execution time. The model follows several execution cycles, performing the scheduled action in each cycle until the plan execution is completed. Initially, the Monitoring module of the execution agent receives the plan Π from the planning agent. Before sending the first action a_1 of Π to execution, the Monitoring performs two operations:

1. It sends Π to the Reactive Planner, which creates, within a time limit, a *repairing structure* for a portion of Π of length l called *plan window*. Thus, l is the

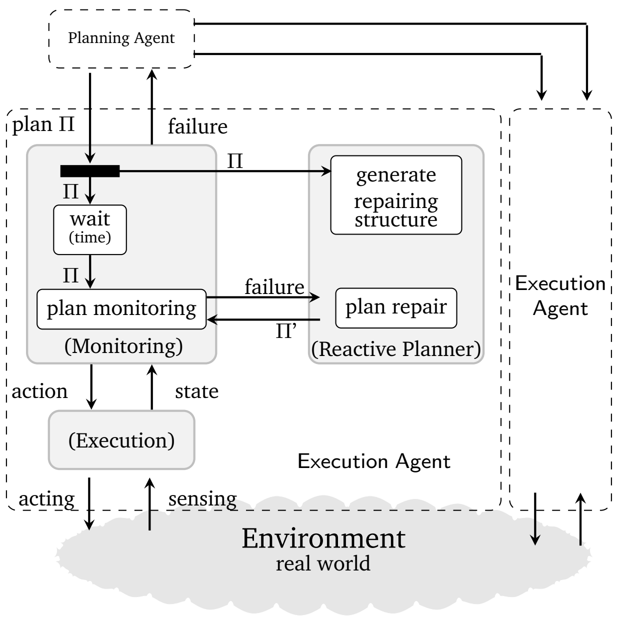

Figure 3.1: Components of the reactive planning and execution architecture and flow of the information.

number of actions or execution cycles of the plan window. The plan window defines the planning horizon the Reactive Planner is going to work with. For example, let's suppose the Reactive Planner creates a repairing structure for a plan window of $l = 3$ ($[a_1, a_2, a_3]$) for plan Π in Figure 3.2. This structure will contain information, in the form of alternative plans, to repair a failure that affects any of these three actions.

2. When the time limit of the Reactive Planner expires, and a repairing structure

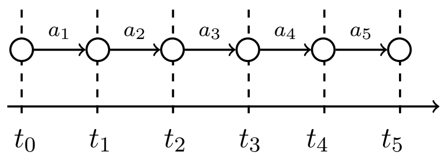

Figure 3.2: Example of a plan Π composed of 5 execution cycles

has been calculated for a particular plan window, the Monitoring checks the variables of the first action of the window; i.e., action a_1 in Figure 3.2. If the sensed values of a_1's variables match the required values for the action to be executed, the Monitoring sends the scheduled action to the Execution module for its execution. Otherwise, a failure is detected and the Monitoring calls the Reactive Planner, which will make use of the repairing structure to fix the failing action.

The Monitoring receives the result of the sensing task from the Execution after the action has been executed, it updates the plan window accordingly by eliminating the already executed action and proceeds with the next action of the plan window. Thus, assuming a_1 was successfully executed, the plan window will be updated to $[a_2, a_3]$, and the Monitoring proceeds with the next action a_2, and so on. If a failure is detected when monitoring a_2 or a_3, the Monitoring will call the Reactive Planner, which will use the same repairing structure to calculate a plan that repairs the failure. This new plan is then concatenated with the rest of actions of Π ($[a_4$, $a_5]$).

The Reactive Planner is generating the repairing structure for the next plan window while the Execution is executing the actions in the previous plan window. Thus, while the system is executing $[a_1, a_2, a_3]$, the Reactive Planner is simultaneously calculating the repairing structure for $[a_4, a_5]$. This ensures there will always be a repairing structure available to attend a failure in the current plan window. The flow goes on as long as there is no more actions to execute.

In summary, the architecture of the RPE model implements the Execution and Monitoring modules of the PELEA architecture inside the execution agent. In addition, we incorporate a Reactive Planner module that endows the agent with fast responsiveness to recover from plan failures.

3.2 Planetary Mars scenario

This section introduces a scenario that we will use as an illustrative and running example throughout this document to help explain various concepts. The example describes a `Mars` domain scenario where a Rover has the mission to analyze some science targets (rock or soil samples) located in different locations (waypoints) and to communicate the results of the analysis to a Lander. The Lander is a robot with a transmission device that allows to communicate faster the results to the Earth. In other words, the Lander establishes a communication bridge between the Rover and the Earth. In addition, the Rover can only communicate with the Lander if the transmission device is located in the Rover and, likewise, it can only analyze science targets if the instrument to analyze samples is operational.

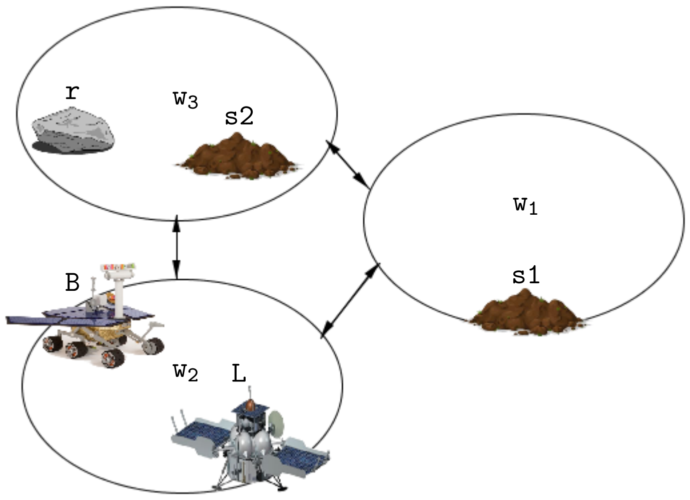

Figure 3.3: Initial state of the Mars domain single motivation scenario.

The illustrative example, depicted in Figure 3.3, describes the initial situation of a particular problem of the `Mars` domain scenario with one rover `B`; three waypoints $\mathtt{w}_1$, $\mathtt{w}_2$, and $\mathtt{w}_3$; one rock sample `r`; two soil samples, `s1` and `s2`; and a Lander `L` which sends the results to the Earth. The waypoint $\mathtt{w}_2$ is the initial location of `L` and `B`. `L` remains always in $\mathtt{w}_2$. The rock sample `r` is located in $\mathtt{w}_3$; `s1` and `s2` are located in $\texttt{w}_1$ and $\texttt{w}_3$, respectively. The mission of B (goal of the problem) is to analyze the soil sample s1, communicate from $\texttt{w}_1$ the results to L, and navigate to $\texttt{w}_2$. The rover B has different capabilities to achieve its mission: navigate from one waypoint to another waypoint, analyze rock or soil samples, and communicate the results of the analysis from the specific waypoint to the lander L. Finally, the rover has good maps to travel between two waypoints.

3.3 Planning concepts

In this section, we formalize the planning concepts necessary to define a planning task and the structures to store the components of a plan [78].

3.3.1 Planning task

Our planning formalism is based on a multi-valued state-variable representation where each variable is assigned a value from a multiple value domain (finite domain of a variable). We use the multi-value variable representation introduced in the most recent version (3.1) of the *Planning Domain Definition Language* (PDDL) [41][1], although we do not make use of the rest of features of PDDL3.1, namely, temporal information, preferences or constraints.

The multi-value variable representation defines a finite set of **variables** $\mathcal{V}$, each associated to a finite domain, $\mathcal{D}_v$, of mutually exclusive values. A variable $v \in \mathcal{V}$ is composed of a set of objects that usually represents the property or attribute of an object of the planning task. For instance, in our Mars scenario, we denote the variable *location of rover* B as loc-B[2], whose domain of possible values is the set of waypoints where B can be placed; that is, $\mathcal{D}_{\texttt{loc-B}} = \{\texttt{w}_1, \texttt{w}_2, \texttt{w}_3\}$. Additionally, we know that rover B may have maps to travel from $\texttt{w}_1$ to $\texttt{w}_2$. We define the

[1]PDDL syntax definition introduced in 2008 by M. Helmert (http://ipc.informatik.uni-freiburg.de/PddlExtension/).

[2]In PDDL3.1, this variable is represented as (loc B). For sake of simplicity, we will represent a variable as a compound element with two or more components joined by a hyphen; e.g., loc-B.

boolean variable *map to travel from* $\texttt{w}_1$ *to* $\texttt{w}_2$ as $\texttt{link-w}_1\texttt{-w}_2$, whose value domain is $\mathcal{D}_{\texttt{link}-\texttt{w}_1-\texttt{w}_2} = \{\texttt{true}, \texttt{false}\}$.

A variable assignment or object fluent (or just **fluent**) is a function f on a variable v such that $f(v) \in \mathcal{D}_v$, wherever $f(v)$ is defined; i.e. a fluent maps a variable v (or tuple of objects) to a value d of its associated domain $\mathcal{D}_v$. We represent a fluent as a tuple $\langle v, d\rangle$, meaning that the variable v takes the value d. For example, the fluent $\langle\texttt{loc-B,w}_2\rangle$ indicates that rover B is in the waypoint $\texttt{w}_2$[3], or the fluent $\langle\texttt{link-w}_1\texttt{-w}_2\texttt{,true}\rangle$ indicates that B has good maps to travel from $\texttt{w}_1$ to $\texttt{w}_2$.

A total variable assignment or **state** applies the function f to all variables in $\mathcal{V}$. A state is always interpreted as a *complete world state*. Appendix A.1.3 shows the description of the state corresponding to the initial situation of our Mars example.

A partial variable assignment or **partial state** over $\mathcal{V}$ applies the function f to some subset of $\mathcal{V}$. The goal of a problem is described as a partial state. In our scenario, the goal of the problem is $\{\langle\texttt{com-s1-w}_1\texttt{,true}\rangle,\langle\texttt{loc-B,w}_2\rangle\}$, where the first fluent denotes the successful communication of the analysis of sample s1 from $\texttt{w}_1$ to the lander L[4].

Definition 3.1 (Planning task). *A planning task is defined as a 4-tuple* $\mathcal{P} = \langle\mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G}\rangle$*:*

- $\mathcal{V}$ *is the finite set of* ***state variables****.*
- $\mathcal{A}$ *is a finite set of* ***actions*** *over* $\mathcal{V}$*. An action* a *is defined as a partial variable assignment pair* $a = \langle$*pre,eff*$\rangle$ *over* $\mathcal{V}$ *called* ***preconditions*** *and* ***effects****, respectively. A precondition is specified as a tuple* $\langle v, d\rangle$*, and an effect is represented as* $\langle v, d'\rangle$*, meaning that* v *changes its value to* d' *whenever the action is executed.*

[3]The same fluent is represented as (= (loc B) $\texttt{w}_2$) in PDDL3.1.

[4]If the communication fails, the fluent would be $\langle\texttt{com-s1-w}_1\texttt{,false}\rangle$

- *$\mathcal{I}$ is a state that represents the **initial state** of the planning task*
- *$\mathcal{G}$ is a partial state over $\mathcal{V}$ called the **goal state** of the planning task.*

An action plan, Π, that solves a planning task $\mathcal{P}$ is a sequence of actions Π $=[a_1, ..., a_n]$ that applied in the initial state $\mathcal{I}$ satisfies the goal state $\mathcal{G}$. The collection of actions in Π is organized in execution cycles or time steps such that one action is executed at a given time. Hence, Π $=[a_1, \ldots, a_n]$ is an ordered sequence of actions where each $a_i \subseteq \mathcal{A}$ is the action to be executed at time step $i-1$ of the plan.

Figure 5.5 shows a solution plan for the planning task mission of rover B. The solution plan has four actions which, if successfully executed, will reach the task goals.

$$\Pi \begin{array}{c|l} a_1 & \text{(Navigate B } \mathtt{w}_2\ \mathtt{w}_1\text{)} \\ a_2 & \text{(Analyze B } \mathtt{s1}\ \mathtt{w}_1\text{)} \\ a_3 & \text{(Communicate B } \mathtt{s1}\ \mathtt{L}\ \mathtt{w}_1\ \mathtt{w}_2\text{)} \\ a_4 & \text{(Navigate B } \mathtt{w}_1\ \mathtt{w}_2\text{)} \end{array}$$

Figure 3.4: Solution plan of the rover B in the Mars scenario.

An action $a_i \in \Pi$ is applicable or executable in a world state S if the fluents contained in S satisfy the preconditions of a_i. More formally: $\forall\langle v, d\rangle \in pre(a_i), \langle v, d\rangle \in S$. When an action a_i is executed, we update the value of a variable v by applying the function $\rho(v, a_i)$, defined as:

$$\rho(v, a_i) := \begin{cases} d & : \quad \exists\langle v, d'\rangle \in eff(a_i) \\ f(v) & : \quad \text{otherwise} \end{cases}$$

The function ρ changes the value of v to d if a_i has an effect $\langle v, d'\rangle$ [5]. Otherwise, the value of v remains unchanged. The result of executing a_i in a state S is a new

[5] In PDDL3.1, preconditions are represented with an **equal** sign (=) and the effects with an **assign** command as can be seen in Appendix A.1.1

state S' that contains the fluents $\langle v, d\rangle$ of S which are not updated by the function $\rho(v, a_i)$ plus the set of new fluents as specified in $eff(a_i)$. That is, the result of applying (the action sequence consisting of) a single action a_i to a state S is:

$$S' := result(S, [a_i]) := \begin{cases} \rho(v, a_i), \forall v \in S & : \text{ if } pre(a_i) \subseteq S \\ undefined & : \text{ otherwise} \end{cases} \tag{3.3.1}$$

The Equation 3.3.1 defines $result$, the *state transition function* from a state S to another state S' after the successful execution of a_i. Accordingly, a plan Π can also be viewed as an ordered sequence of states.

Definition 3.2 (Solution plan as an ordered sequence of states). *A solution plan* $\Pi = [a_1, \ldots, a_n]$ *for a planning task* $\mathcal{P}$ *is as a chronologically ordered sequence of states* $[S_0, S_1, \ldots, S_n]$*, where:*

- $S_0 := \mathcal{I}$
- $\mathcal{G} \subseteq S_n$
- $S_i := result(S_{i-1}, [a_i])$

Executing Π in the initial state $\mathcal{I}$ results in a sequence of states $[S_1, ..., S_n]$ such that S_1 is the result of applying a_1 in $\mathcal{I}$, S_2 is the result of applying a_2 in S_1, and S_n is the result of applying a_n in S_{n-1}. A plan Π is a solution plan iff $\mathcal{G} \subseteq S_n$. The result of executing Π in a state S can also be recursively defined by $result(S,[a_1, \ldots, a_n]) = result(result(S,[a_1, \ldots, a_{n-1}]), [a_n])$.

3.3.2 Representing plan structures

Definition 3.2 views a plan as the result of the successive execution of the actions of Π in the initial state $\mathcal{I}$ of the problem. This is the usual way of interpreting a plan in classical planning [78], defining the *resulting states* after the execution of the actions at each time step in the plan.

Additionally, we can also interpret a plan from the point of view of the world conditions (fluents) that are necessary for the plan to be executed. That is, instead of viewing a plan as the result of its execution, we can view a plan as the *necessary conditions* for the execution of the plan actions. A plan can thus also be defined as a sequence of partial states, rather than world states, which contain the *minimal set of fluents* that must hold in the world for executing each action of the plan.

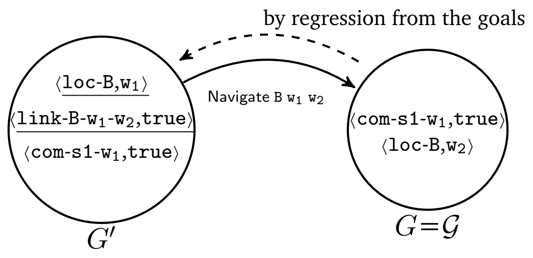

Figure 3.5: Plan as a sequence of partial states.

Figure 3.5 depicts the last action of the solution plan shown in Figure 5.5 (Appendix A.1.2 describes the meaning of each variable). G is the goal state $\mathcal{G}$ of the planning task $\mathcal{P}$, a partial state that contains two fluents $\{\langle \texttt{com-s1-w}_1\texttt{,true}\rangle, \langle \texttt{loc-B,w}_2\rangle\}$. The last action of the plan is the action (Navigate B $\texttt{w}_1$ $\texttt{w}_2$) with preconditions $\{\langle \texttt{loc-B,w}_1\rangle, \langle \texttt{link-B-w}_1\texttt{-w}_2\texttt{,true}\rangle\}$ and effects $\{\langle \texttt{loc-B,w}_2\rangle\}$. Then, the fluents needed to be able to execute the action and achieve the fluents in $\mathcal{G}$ are represented in state G'. We can observe that G' does not only contain the fluents that match the preconditions of the action Navigate but also the fluent $\langle \texttt{com-s1-w}_1\texttt{,true}\rangle$. This fluent is a goal of $\mathcal{G}$ that is not achieved by the effects of the action `Navigate` but at some point earlier in the plan and thereby, $\langle \texttt{com-s1-w}_1\texttt{,true}\rangle$ must hold in G' in order to guarantee that it is satisfied in $\mathcal{G}$.

The state G' in Figure 3.5 is called a ***regressed*** *partial state* because it is calculated by regressing the goals in $\mathcal{G}$ through the action Navigate. The notion of a regressed partial state is inspired by the STRIPS triangle table introduced by

PLANEX [36], a system that was designed as both a plan executor and a plan monitor for Shakey, the Stanford Research Institute robot system. PLANEX was conceived as a system responsible for monitoring the execution of a plan and supervising the execution of a sequence of actions.

The core idea of PLANEX is to represent plans in a way that supports monitoring by representing the plan structure in a triangle table that stores a generalized plan. Specifically, the idea consists in annotating plans with sufficient and necessary conditions that can be checked at execution time to confirm the validity of a plan. These conditions correspond to the regression of a fluent f through an action a, which determine the fluents that must hold prior to a being executed if and only if f holds after a is executed [88]. Roughly, the regression of f through an action a is a sufficient and necessary condition for the satisfaction of f following the execution of a.

Regression is a fundamental tool in automated planning, in which we seek the conditions needed to reach a state, achieve a fact or perform an action. In the context of classical planning, the work of Rintanen in [91] describes how to compute the regression of a formula ϕ through an action a. Rintanen introduces a full regression definition which is not needed in our case because (1) we assume a restricted PDDL syntax (no conditional effects or disjunction of preconditions in the actions) and (2) we already dispose of the sequence of actions of the solution plan so we only need to regress through these actions. Following Rintanen's definition of regression, we present a simplified definition of regressing a set of fluents through an action.

Let a be an action and G a goal state such that $eff(a) \subseteq G$. The *partial state* G', in which a is executable, is calculated by the *regressed transition function* $regress(G, a)$, defined as:

$$G' := regress(G, a) := G \setminus eff(a) \cup pre(a) \tag{3.3.2}$$

G' is a partial state that represents the minimal set of fluents that must hold in a world state in order to generate G by means of the execution of a. Notice that G' includes $pre(a)$ plus the fluents of G that are not produced by $eff(a)$ ($G \setminus eff(a)$). This is the set of fluents that are achieved before G' in the plan and must keep their values until G.

This regression computational mechanism has been widely used to monitoring plan validity and plan optimality during execution [44] as well as in reactive planning [80].

Now, we can define a plan as a sequence of partial states via the regressed transition function $regress$.

Definition 3.3 (Solution plan as an ordered sequence of partial states). *Given a solution plan* $\Pi = [a_1, \ldots, a_n]$ *for a planning task* $\mathcal{P}$*, the* regressed plan *for* Π *is defined as a chronologically ordered sequence of partial states* $[G_0, G_1, \ldots, G_n]$*, where:*

- $G_0 \subseteq \mathcal{I}$
- $G_n := \mathcal{G}$
- $G_{i-1} := regress(G_i, a_i)$

A regressed plan $[G_0, \ldots, G_n]$ derived from Π denotes the fluents that must hold in the environment at each time step t to successfully execute the actions in Π from t onwards. In other words, a partial state G_i denotes the set of fluents that must hold in the world state S at time $t = i$ ($G_i \subseteq S$) to ensure the sequence of actions $[a_{i+1}, \ldots, a_n]$ is executable in S, thus guaranteeing the goals in $\mathcal{G}$ are achieved. This definition allows us to discern between the fluents that are relevant for the execution of a plan and those ones that are not.

Figure 3.6 shows the regressed plan $[G_0, \ldots, G_4]$ derived from the solution plan shown in Figure 5.5. This plan is calculated by successively applying $regress$ through the actions of Π. For instance, an effect of the action a_2 is the fluent $\langle \texttt{have-B}, \texttt{s1} \rangle$ which appears in G_2; G_1 is the partial state resulting from applying

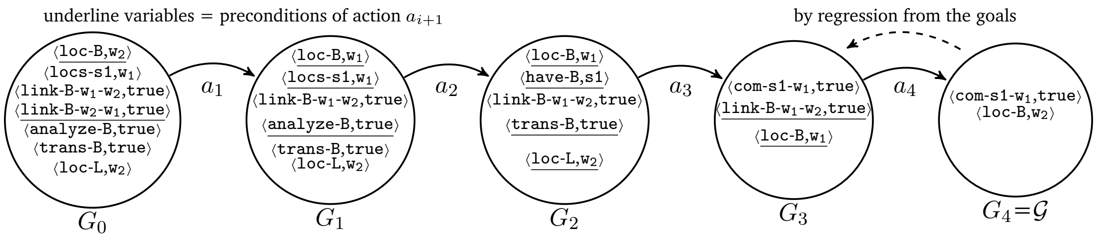

Figure 3.6: Plan as a sequence of partial states for the plan Π of B in the Mars scenario.

$regress(G_2, a_2)$, which includes $pre(a_2)$ (the fluents which are underlined in node G_1) plus the fluents that are in G_2 but are not produced by $eff(a_2)$. Therefore, if the sensor reading returns a world state in which all of the fluents in G_1 hold, then action a_2 is executable in such a world state and accordingly $result(result(S,[a_2]),[a_3, a_4])$; if the fluents in G_2 occur in the subsequent world state then a_3 is executable and therefore $result(result(S,[a_3]),[a_4])$ and so on. The last partial state, G_4, comprises $\mathcal{G}$, the goals of the planning task. In terms of plan monitoring, G_4 represents the fluents that satisfy the preconditions of a fictitious final action, a_f, where $pre(a_f) = \mathcal{G}$ and $eff(a_f) = \emptyset$, i.e. $\mathcal{G}$ is monitored by checking the preconditions of a_f.

3.4 Reactive Planner

In this section, we provide a detailed description of the Reactive Planner module embedded into the RPE model [55, 54]. The design and development of the Reactive Planner is the main novelty of our RPE model.

Firstly, we must note that execution agents internally work with the same PDDL encoding of the planning task, thus avoiding a high-level to a low-level translation of the data. This contrasts with other approaches, specifically devoted to executing real-world applications such as robotics, that require to translate the high-level planning language into a low-level language understandable by the execution agents or robots [39]. Our proposal, however, is a domain-independent model for simulating plan execution and reactive planning when a timely repair is needed. Thereby, in our model, the execution agent is designed to work with the same language of the planning agent, thus facilitating the communication between the agents and avoiding the overhead of translating the data. Nevertheless, if the RPE is intended to be used in a real-world application where execution agents are required to work with low-level data, a high-level to low-level and a low-level to high-level translation modules could be easily incorporated in the architecture without affecting the internal behaviour of the RPE model.

The key concept of the Reactive Planner is the *repairing structure*. Generally speaking, a repairing structure is a search space composed of partial states that encodes recovery plans for potential failures that may occur in a portion of a plan Π, called *plan window*. We use the term plan window equivalently to the concept of planning horizon introduced in IDEA [8, 38]. In practice, IDEA never works with a planning horizon longer than the minimal horizon (one execution cycle) in order to limit the search space of the planner. This grants IDEA more reactivity but also provides less information to repair a failure. In contrast, our Reactive Planner module works with a plan window whose length is delimited by the available time the planner is given to build up the repairing structure.

3.4.1 Construction of a repairing structure

The Reactive Planner is given a finite amount of time to generate the repairing structure. First, it estimates the size of the search space that is capable to build within the available time. Particularly, it estimates the length of the plan window and the maximum depth of the search space (this estimation process is explained in detail in Section 3.4.3.1). Once the length of the plan window (l) and the maximum depth of the search space (m) have been calculated, the Reactive Planner proceeds with the construction of the repairing structure.

Definition 3.4 (Repairing structure). *Given a planning task $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$, and a solution plan Π =[$a_1, \ldots, a_n$] or regressed plan [$G_0, \ldots, G_n$] for $\mathcal{P}$; a repairing structure $\mathcal{T}$ of maximum depth m associated to a plan window of length l of Π ([$G_0, \ldots, G_l$]) is defined as follows* [6]*:*

1. *The root node or partial state at depth level 0 of $\mathcal{T}$ is G_l*
2. *Nodes of $\mathcal{T}$ at depth level $d + 1$ are partial states calculated by applying the regressed transition function $regress$ over the nodes at depth d; and arcs between nodes are actions of the planning task.*
3. $depth(G) <= m, \forall\, G \in \mathcal{T}$.

The repairing structure $\mathcal{T}$ is a **search tree** composed of partial states whose root node is G_l, the last partial state of the regressed plan of the window associated to $\mathcal{T}$. We apply the regressed transition function $regress$ to the root node to generate the nodes G at the next depth level, $depth(G) = 1$, and subsequently to generate the nodes at the next level, $depth(G) = 2$, and so on until we generate the nodes at the maximum depth, $depth(G) = m$. In general, the search space defines a uni-directional *graph* where one same node can be reached through more than one path. We will analyze this feature at the end of this section.

Figure 3.7 shows two repairing structures, $\mathcal{T}_1$ and $\mathcal{T}_2$, for the regressed plan of Figure 3.6. $\mathcal{T}_1$ is the repairing structure associated to the plan window $[a_1, a_2]$, or, equivalently, to the regressed subplan $[G_0, \ldots, G_2]$. The *root node* of $\mathcal{T}_1$ is G_2 and the maximum depth of $\mathcal{T}_1$ is $m = 6$. A branch in $\mathcal{T}_1$ is a sequence of actions or regressed plan that represents a path from any node of $\mathcal{T}$ to the root node G_2.

The process of generating a repairing structure $\mathcal{T}$ is shown in Algorithm 1. It

[6]We refer to the depth d of a node $G \in \mathcal{T}$ as $depth(G)$.

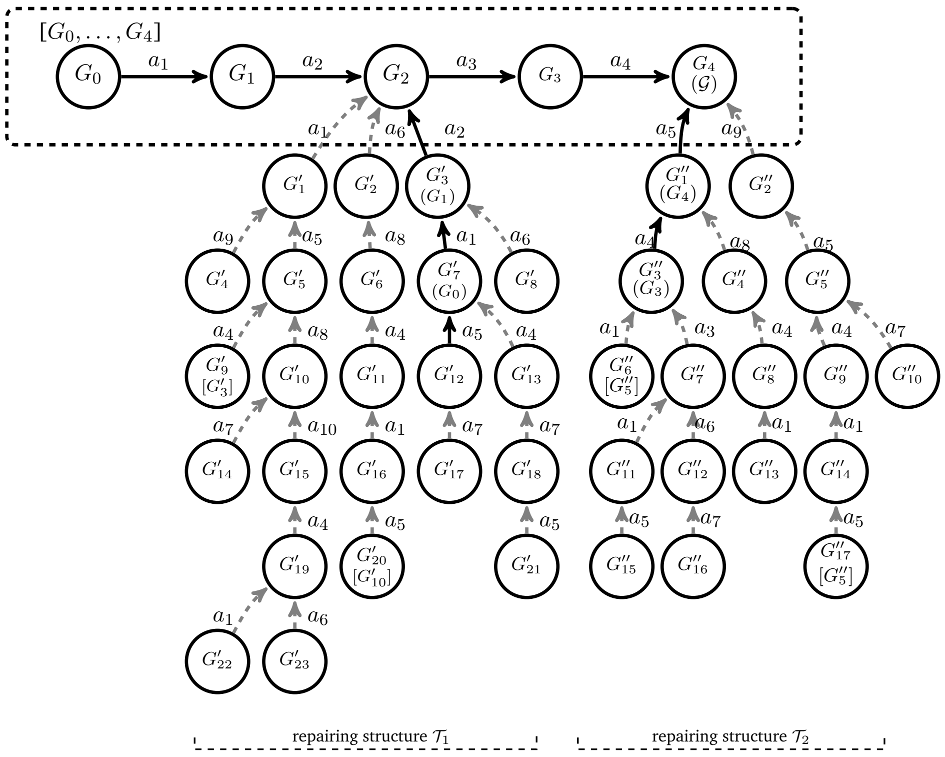

Figure 3.7: Repairing structures for the solution plan of B in the planetary Mars scenario. $[G_0, \ldots, G_4]$: plan as partial states for $[a_1, a_2, a_3, a_4]$, $\mathcal{T}_1$: the repairing structure associated to the actions $[a_1,a_2]$, and $\mathcal{T}_2$: the repairing structure associated to the actions $[a_3,a_4]$.

expands the root node G_l via the application of the regressed transition function $regress(G_l, a)$ following Equation 3.3.2 (line 6 of the algorithm). The algorithm is a classical backward construction of a planning space following a breadth-first search procedure [78], where a node G is expanded until it reaches the maximum depth limit or it satisfies other pruning conditions (see the paragraph related to the repeated states and supersets in this same section). The purpose of Algorithm 1 is to generate all possible sequences of actions of maximum length m that eventually reach G_l.

In Definition 3.3, we saw that a set of fluents G is regressed through the action a of a given plan Π. In Algorithm 1, however, we do not have such a plan Π to regress the fluents of G_l because the purpose of the algorithm is precisely to find all possible sequences of actions that allow to reach G_l. In this case, the operation of regressing the fluents G through an action a of the planning task ($a \in \mathcal{A}$) must check whether a is a **relevant** action to achieve some fluent $f \in G$ or not. An action a is relevant for G and originates an arc (G', G) in $\mathcal{T}$, where $G' = regress(G, a)$, if a does not cause any conflict with the fluents in G and G'. Specifically, finding a relevant action a for a given partial state G requires to check two consistency restrictions: (1) that $eff(a)$ does not conflict with the fluents in G, and (2) that $pre(a)$ does not conflict with the fluents in the regressed goal state G'.

```
Input: Generate_Search_Space($G_l, \mathcal{A}, m$)
 1: $\mathcal{Q} \leftarrow \{G_l\}$
 2: $\mathcal{T} \leftarrow \{G_l\}$
 3: while $\mathcal{Q} \neq \emptyset$ do
 4:    $G \leftarrow$ extract first node from $\mathcal{Q}$
 5:    for all $\{a \in \mathcal{A} \mid relevant(G, a)$ is true$\}$ do
 6:       $G' \leftarrow regress(G, a)$
 7:       if $G' \notin \mathcal{T}$ then
 8:          if $\exists\ G'' \in \mathcal{T} \mid G'' \subset G'$ then
 9:             mark $G'$ as superset of $G''$
10:          else
11:             if $depth(G') < m$ then
12:                $\mathcal{Q} \leftarrow \mathcal{Q} \cup G'$
13:          set an arc $a$ from $G'$ to $G$
14:          $\mathcal{T} \leftarrow \mathcal{T} \cup G'$
15: return $\mathcal{T}$
```

Algorithm 1: Generate a repairing structure $\mathcal{T}$ from a given partial state G_l up to a depth m with a set of actions $\mathcal{A}$.

We define a function $conflict(F_i, F_j)$ that given two any sets of fluents, F_i and F_j, checks whether a conflict between the fluents of F_i and F_j exists or not:

$$conflict(F_i, F_j) := \begin{cases} true & : \ \exists\langle v, d\rangle \in F_i \text{ and } \exists\langle v, d'\rangle \in F_j \text{ and } d \neq d' \\ false & : \ \text{otherwise} \end{cases} \tag{3.4.1}$$

The function $conflict(F_i, F_j)$ holds if it exists a variable v in the two set of fluents F_i and F_j with different values d and d', respectively. For instance, assuming two fluents $\langle \texttt{loc-B},\texttt{w}_2\rangle \in F_i$ and $\langle \texttt{loc-B},\texttt{w}_1\rangle \in F_j$, $conflict(F_i, F_j)$ will return $true$ because the variable `loc-B` has a different value in the two sets of fluents.

Now, we provide a definition of a relevant action a for a partial state G:

$$relevant(G, a) := \begin{cases} true: & ((\exists\langle v, d'\rangle \in G \text{ and } \exists\langle v, d'\rangle \in eff(a)) \text{ and} \\ & (\neg(conflict(eff(a), G))) \text{ and} \\ & (\neg(conflict(pre(a), regress(G, a))))) \\ & \\ false: & \text{otherwise} \end{cases} \tag{3.4.2}$$

The function **relevant**(G, a) in Equation 3.4.2, returns **true** if the action a is relevant for a node G; i.e., if a achieves a fluent $\langle v, d'\rangle$ in G and the rest of effects of a do not conflict with the fluents in G; and $pre(a)$ does not cause a conflict with the fluents of the regressed partial state $regress(G, a)$. The relevant function is applied to every node in $\mathcal{T}$ and for each action in the planning task (line 5 of Algorithm 1). For example, in $\mathcal{T}_1$ of Figure 3.7, the actions a_1, a_2 and a_6 are relevant actions for the root node G_2 of $\mathcal{T}_1$ because:

1. the effects of these actions generate a fluent of G_2 and
2. the rest of effects of the actions do not conflict with the fluents of G_2 and
3. the regressed partial states $G'_1 = regress(G_2, a_1)$, $G'_2 = regress(G_2, a_2)$ and $G'_3 = regress(G_2, a_6)$ do not conflict with $pre(a_1)$, $pre(a_2)$ and $pre(a_6)$, respectively.

Hence, the construction of $\mathcal{T}$ follows Algorithm 1, a goal-regression algorithm that applies $regress(G, a)$ for each relevant action a in a partial state G which has not reached the maximum depth m yet (lines 4-6 of Algorithm 1). Note that the arcs in a repairing structure are displayed in the opposite direction of the node generation because we want to point out the order of application of the actions (see Figure 3.7).

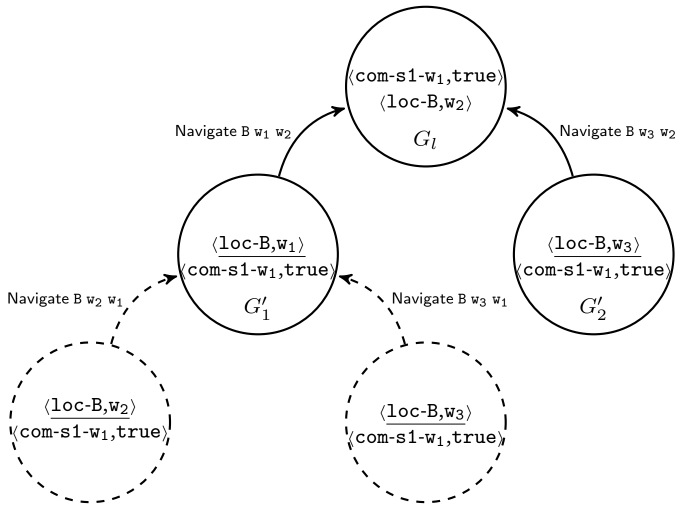

Figure 3.8: Repeated states with reversible actions and redundant paths.

3.4.1.1 Repeated states and supersets.

The search space $\mathcal{T}$ is actually a graph due to the existence of repeated states, i.e. multiple paths that reach one same partial state during the construction of $\mathcal{T}$. More specifically, repeated states are originated as a consequence of redundant paths (e.g. the application of various actions in either order) and reversible actions. Figure 3.8 shows a simple example of reversible actions and redundant paths. It shows a tree with three partial states G_l, G'_1 and G'_2 generated by the actions (Navigate B w_1 w_2) and (Navigate B w_3 w_2), respectively. For simplicity, in this example, the actions Navigate have as precondition only the location of the rover B. The node G'_1 has two dashed child nodes which are not inserted in the tree. The left dashed node is a repeated state of G_l, which is generated by the application of two reversible actions (Navigate B w_1 w_2) and (Navigate B w_2 w_1). The right dashed node is a repeated state of G'_2 because both contain the same fluents and the path to reach G_l is shorter from G'_2. Repeated states are not inserted in the tree.

Another type of nodes that can be found in a repairing structure are *superset* nodes. More formally, a node $G'_j \in \mathcal{T}$ is said to be a superset of a node G'_i if $G'_i \subset G'_j$. Superset nodes are inserted in the tree but they are not expanded. If G'_j is a superset of a node G'_i which is located at an upper depth level, then G'_i already comprises the minimal set of fluents that must hold in the world to be able to reach the root node. Therefore, superset nodes do not bring any new data with respect to their subset nodes because if the fluents of superset node G'_j hold in the world state S ($G'_j \subseteq S$), it is obviously the case that $G'_i \subseteq S$, too, and the path from the subset node G'_i to the root node is shorter than the path from G'_j to the root node. Superset nodes are marked as such and inserted in the tree but they are not expanded. For example, in $\mathcal{T}_1$ of Figure 3.7, the node G'_9 is a superset of node G'_3 [7] ($G'_3 \subset G'_9$). G'_3 is a partial state where the rover `B` and the soil sample are located in the same location `w1` and the action (a_2) to analyze the soil is the shortest path to reach the root node G_2.

Algorithm 1 makes two nodes in $\mathcal{T}$ be connected by a single path. Since we are interested in keeping only the shortest (optimal) path between any pair of nodes, the construction of $\mathcal{T}$ neglects repeated states (line 7 in Algorithm 1) and avoids the expansion of *superset* nodes (lines 8 and 9). These two pruning techniques allow us to encode the optimal path between each pair of nodes.

3.4.1.2 Complexity of the construction algorithm.

Another relevant issue in the construction of $\mathcal{T}$ is that a set of variables induce a state space that has a size that is exponential in the set, and, for this reason, planning, as well as many search problems, suffer from a combinatorial explosion. Even though nodes in $\mathcal{T}$ are partial states that contain far less fluents than world states, the large

[7]The subset node is represented between **brackets** []

size of the repairing structures are sometimes unaffordable for a reactive system. With the aim of reducing the size of $\mathcal{T}$, when expanding a node G' we only consider the actions of the planning task that modify a fluent $\langle v, d' \rangle$ of G' and whose variable v is a *relevant variable*; that is, a variable which is involved in the preconditions or effects of the actions of the plan window associated to $\mathcal{T}$. For instance, given the repairing structure $\mathcal{T}_1$ associated to the plan window $[a_1, a_2]$ or regressed plan $[G_0, G_1, G_2]$ in Figure 3.7 and any node $G' \in \mathcal{T}_1$, the only actions which are applied in G' are those a such that $\langle v, d' \rangle \in (eff(a) \cap G')$ and v is a relevant variable involved in the set $pre(a_1) \cup eff(a_1) \cup pre(a_2) \cup eff(a_2)$. Actually, this set contains the fluents that may need to be repaired when a failure occurs.

On the other hand, note that in every repairing structure $\mathcal{T}$ there always exists a branch or part of a branch which arcs are labeled with exactly the same actions of the plan window associated to $\mathcal{T}$. In Figure 3.7, we can observe that the first arc of the rightmost branch is action a_2 and the subsequent arc is labeled with action a_1. Therefore, the partial states G'_3 and G'_7 are the same as G_1 and G_0, respectively.

The time complexity of the Algorithm 1 responds to the classical complexity of the generation of a tree, that is $\mathcal{O}(b^m)$, where b is the branching factor or maximum number of successors of any node in the tree.

3.4.2 Finding a recovery plan in a repairing structure

A path or branch in a repairing structure $\mathcal{T}$ associated to a plan window $[G_0, \ldots, G_n]$ represents a (regressed) plan to reach the root node G_l. Specifically, a path is interpreted as a recovery plan that leads the current world state to another state from which the execution of the plan Π can be resumed. Given the current world state S, the idea is to find a partial state G' in the tree $\mathcal{T}$ such that $G' \subseteq S$. Then, the path from G' to G_l is the sequence of actions that when executed in S will result in a world state S' such that $G_l \subseteq S'$; consequently, the execution of Π can be resumed from G_l onwards. In Figure 3.7, assuming that S is the current world state and action a_1 of Π fails (a_1 being comprised in the plan window $[a_1, a_2]$ associated to $\mathcal{T}_1$), the Reactive Planner will use $\mathcal{T}_1$ to find a recovery path. Let's suppose that $G'_{16} \subseteq S$; applying the plan $[a_1, a_4, a_8, a_6]$ in S will reach the root node G_2, from which the rest of Π, $[a_3, a_4]$, can be executed; or, if it were the case that $G'_{18} \subseteq S$, the application of the sequence of actions $[a_7, a_4, a_1, a_2]$ in S will reach the root node G_2, from which the rest of Π, $[a_3, a_4]$, can be executed. Hence, all recovery plans in $\mathcal{T}_1$ have one thing in common: they eventually guide the execution of the plan towards G_2, the root node of $\mathcal{T}_1$.

In the following, we will thoroughly explain the process to find a recovery path when a failure occurs during the execution of a plan and we will show how this process applies to our Mars scenario.

3.4.2.1 Iterative recovery process

Let $\Pi=[a_1, \ldots, a_n]$ be the plan under execution and $[a_i, \ldots, a_l]$ or, equivalently, $[G_{i-1}, \ldots, G_l]$, the plan window of Π associated to a repairing structure $\mathcal{T}$; and let us also suppose that the preconditions of a_i do not hold at the time of execution when the current world state is S. The Reactive Planner applies an iterative process over the partial states of the plan window $[G_{i-1}, \ldots, G_l]$ until a path from one of these partial states to a node $G' \subseteq S$ is found in the tree $\mathcal{T}$. Specifically, at each iteration, the process works with one partial state of $G_{i-1}, \ldots, G_l$, finds the corresponding partial state in $\mathcal{T}$ (we will refer to this node in the tree as G'_t) and then it goes all the way down through the descendents of G'_t until it finds a node G' that satisfies $G' \subseteq S$, if possible:

1. if $G'_t=G_{i-1}$ and a descendant node G' of G'_t is found, then applying the path from G' to G'_t in S will lead a new state S' in which a_i is executable; otherwise
2. if $G'_t=G_i$ and a descendant node G' of G'_t is found, then applying the path from G' to G'_t in S will lead a new state S' in which a_{i+1} is executable; otherwise

3. continue until G'_t=G_l

Hence, the idea is to find a node $G' \in S$ in the repairing structure such that the application of the sequence actions between G' and the root node will eventually reach a node which is equal to the partial state G_{i-1} of the plan window in which a_i is executable; or a node equal to G_i in which a_{i+1} is executable; or, ultimately, a node equal to G_l, in which case the recovery path from S to reach the root node G_l is a path which does not contain any of the actions of the plan window.

The iterative process to find a recovery path is shown in Algorithm 2. When the action a_i of the plan window $[a_i, \ldots, a_l]$ fails, Algorithm 2 is activated with the repairing structure $\mathcal{T}$, the regressed plan $[G_{i-1}, \ldots, G_l]$ associated to $\mathcal{T}$ and the set of fluents of the current world state S. The algorithm successively iterates over the partial states in $G_{i-1}, \ldots, G_l$, finds such a state in the tree (G'_t) and then tries to find a node $G' \subseteq S$ from G'_t [8]. The process stops when some G' is found from any of the partial states of the plan window, in which case the path from G' to G'_t is concatenated with the plan from G'_t to the root node G_l. If no G' is found for any state of the plan window, this means that $\mathcal{T}$ does not comprise the necessary information to find a recovery path, in which case the planning agent is invoked to perform a replanning task that calculates a new plan from scratch.

```
Input: Iterative_Search_Plan($\mathcal{T}$, $S$, $[G_{i-1}, \ldots, G_l]$ )
1: Π' ← $\emptyset$
2: $\mathcal{Q} \leftarrow \{G_{i-1}, \ldots, G_l\}$
3: while $\mathcal{Q} \neq \emptyset$ and Π' $= \emptyset$ do
4:     $n$ ← pop node from $\mathcal{Q}$
5:     $G'_t$ ← find $n$ in $\mathcal{T}$
6:     if $\exists$ descendant $G'$ of $G'_t \mid G' \subseteq S$ then
7:         Π' ← concatenate path from $G'$ to $G'_t$ to the plan from $G'_t$ to $G_l$
8: return Π'
```

Algorithm 2: Iterative algorithm to find a recovery path

[8]The algorithm applies a modified breadth-first search that prunes the already descendant nodes in subsequent iterations.

3.4.2.2 Working on the Mars scenario

Let's see now how the iterative process applies to our Mars scenario. Consider the plan $\Pi = [a_1, a_2, a_3, a_4]$ of Figure 5.5, the plan window $[a_1, a_2]$ associated to $\mathcal{T}_1$ in Figure 3.7 ($[G_0, \ldots, G_2]$) and that the preconditions of $a_1 =$(Navigate `B` $\mathtt{w}_2$ $\mathtt{w}_1$) are not satisfied in S. Notice that a failure that occurs in the first action of a plan is always due to an exogenous event and not to an erroneous execution of the preceding action in the plan. Let's assume a_1 is not executable because a windstorm hits the rover producing an unexpected wrong location of rover `B`, no longer being at $\mathtt{w}_2$ but at $\mathtt{w}_3$.

- First iteration (G_0). Algorithm 2 finds G_0 in $\mathcal{T}_1$ ($G_0 = G'_7$ in $\mathcal{T}_1$) and searches for a descendent node G' of G'_7 that satisfies $G' \subseteq S$. The nodes that are reachable from G'_7 are G'_{12}, G'_{13}, G'_{17}, G'_{18} and G'_{21}. If none of these states match the current world state S, that is $G'_{12} \nsubseteq S$ and $G'_{13} \nsubseteq S$ and $G'_{17} \nsubseteq S$ and $G'_{18} \nsubseteq S$ and $G'_{21} \nsubseteq S$, then it means no plan repair actually exists to reach G'_7 from S in $\mathcal{T}_1$, and, hence, there is no way to get rover `B` back to $\mathtt{w}_2$ from $\mathtt{w}_3$.

- Second iteration (G_1). Assuming there is no solution to move rover `B` back to $\mathtt{w}_2$, the Reactive Planner locates the next partial state G_1 in the tree (node $G_1 = G'_3$ in $\mathcal{T}_1$), a node in which rover `B` is in $\mathtt{w}_1$ in order to analyze the soil. Hence, for every descendant node of G'_3 (excluding G'_7 and its descendant nodes which were already explored in the previous iteration), the Reactive Planner checks whether it exists $G' \subseteq S$, in which case the Reactive Planner will return the plan from G' to G'_3. Let's assume that $G' = G'_8$ so a path that reaches G'_3 from S actually exists. This path comprises the action a_6, (Navigate `B` $\mathtt{w}_3$ $\mathtt{w}_1$), which moves rover `B` from $\mathtt{w}_3$ to $\mathtt{w}_1$. Then, the algorithm returns the recovery plan $[a_6]$, concatenated with $[a_2]$ (analyzing the soil sample) and the Reactive Planner will then append the rest of the actions of Π ($[a_3, a_4]$).

3.4.2.3 Discussion

Two issues related to for finding a recovery path are worth mentioning here. First, it is important to highlight that the choices to find a plan increase when the partial state to reach is closer to the root node of the tree. This can be graphically observed in Figure 3.9. The left figure shows a tree $\mathcal{T}$ of depth m associated to a plan window of length $l = 3$. The first iteration of Algorithm 2 is aimed at finding a node G' that reaches a node equal to G_0. In such a case, actions a_1, a_2 and a_3 will be included in the recovery plan, and consequently, Algorithm 2 restricts the space to search to h levels of the tree, the shadowed portion of the tree in the figure. However, the objective of the second iteration of Algorithm 2 is to find a path that reaches a node equal to G_1 (right figure), in which case the recovery path will only comprise the actions a_2 and a_3 of the plan window, and choices to find a state that matches S increase as well as the choices of finding a recovery path to reach G_1. In conclusion, the farther the partial state of the plan window from the root node, the fewer plan repair alternatives but, however, the recovery plan guarantees more stability with respect to the original plan because it will include more actions of the plan window. In contrast, if the partial state is closer to the root node then the search space to find a recovery path is bigger so there are more choices to find it although the plan found might not keep any of the actions of the plan window. Clearly, the more flexibility, the less stability.

Secondly, it might happen that no recovery plan at all is found. Notice that the tree has a limited size in terms of l and m, which is determined by the available time to build the tree, and the information included in the tree may not be sufficient to solve all the potential contingencies. The principle underpinning reactive systems is that of providing a prompt reply, and this requires to work with bounded data structures. Moreover, reactive responses are applied when slight deviations from the main course of actions occur. A major fault that makes a variable take a value that is not considered in the repairing structure would likely need a more deliberative

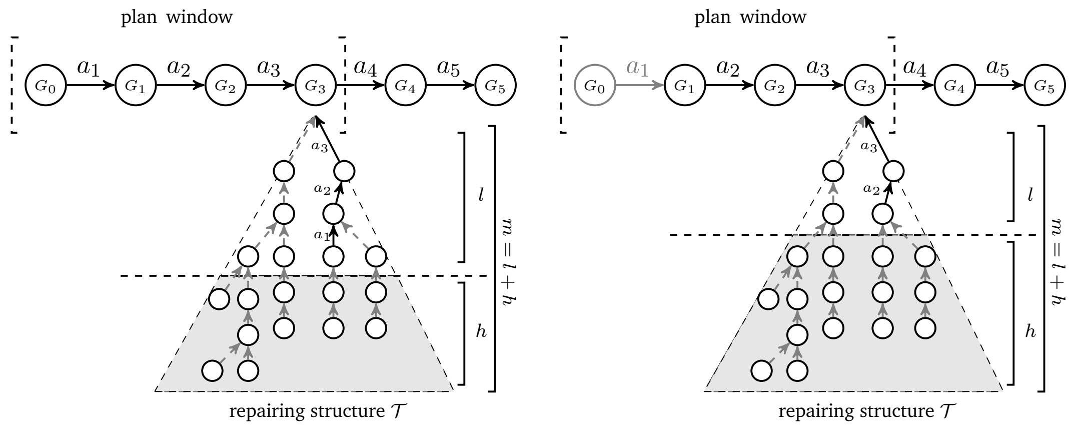

Figure 3.9: Repairing structures abstract.

response.

In general, the higher the value of the plan window length l, the more partial states or choices to find a recovery plan; and the deeper the tree, the longer the recovery plans comprised in $\mathcal{T}$. The minimum value of m must be $l+1$ in order to ensure that the tree comprises at least one action that repairs the first action of the plan window associated to $\mathcal{T}$. Finally, since nodes in the search tree are partial states, our RPE model only handles the minimal data set that is necessary to carry out a repairing.

3.4.3 Calculating repairing structures for a solution plan

The number of repairing structures necessary to keep track of the execution of a plan Π depends on the available time the Reactive Planner has to create the search trees, which, in turn, delimits the size of the trees. The size of a repairing structure $\mathcal{T}$ is subject to two parameters, the plan window length l associated to $\mathcal{T}$, and the maximum depth limit m of the search tree. For example, in Figure 3.7, the length of the plan window of $\mathcal{T}_1$ is $l = 2$ and the depth limit of $\mathcal{T}_1$ is $m = 6$, values which result from the time limit the Reactive Planner is given to build $\mathcal{T}_1$; as for $\mathcal{T}_2$, the length of the plan window is $l = 2$ and maximum depth $m = 5$, values which likewise depend on the available time the Reactive Planner has to build $\mathcal{T}_2$.

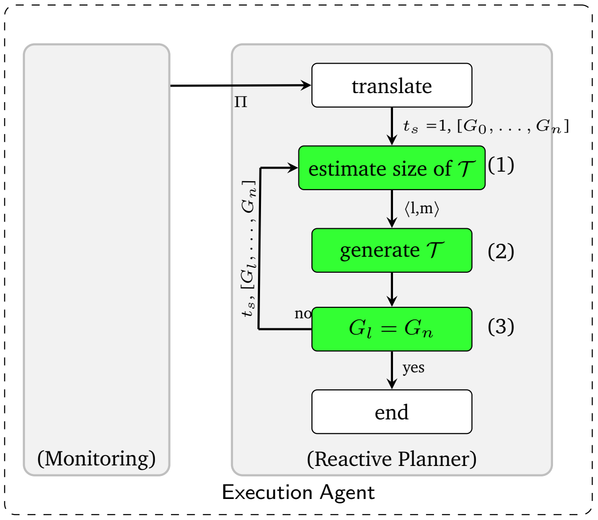

Figure 3.10: Iterative time-bounded construction to generate repairing structures

The process to generate the various repairing structures for a plan Π is an iterative time-bounded construction process composed of three steps as depicted in Figure 3.10 (green boxes). When the Reactive Planner receives the initial plan Π from the Monitoring module, it translates Π into a sequence of partial states $[G_0, \ldots, G_n]$ using the Definition 3.3 and starts the iterative time-bounded process:

- **First iteration $[G_0, \ldots, G_n]$:** In the first iteration, the Reactive Planner is given a time limit of one execution cycle; i.e., the value of t_s is equal to one cycle time. Then, it performs the following steps:

    1. Estimates the values of l and m for $\mathcal{T}_1$ (the process to estimate the size of a repairing structure is explained in Section 3.4.3.1).
    2. It generates $\mathcal{T}_1$ for the plan window $[G_0, \ldots, G_l]$ with maximum depth m as explained in Section 3.4.1.
    3. If $G_l \neq G_n$ then at least a second repairing structure is needed to cover the entire plan Π. In this case, the Reactive Planner proceeds with the generation of $\mathcal{T}_2$.

- **Subsequent iterations [$G_l, \ldots, G_n$]:** The available time for building the second and subsequent repairing structures is subject to the time the Execution will need to execute the actions in the preceding window. Specifically, the value of t_s for building a repairing structure $\mathcal{T}_i$ other than $\mathcal{T}_1$ is set equal the time of as many execution cycles as number of actions covered by the preceding repairing structure $\mathcal{T}_{i-1}$. Thus, the Reactive Planner:

  1. Estimates the size l' and m' of $\mathcal{T}_2$.
  2. Generates $\mathcal{T}_2$ for $[G_l, \ldots, G_{l+l'}]$ with maximum depth m';
  3. If $G_{l+l'} \neq G_n$ then the Reactive Planner proceeds with the generation of the next repairing structure for the remaining actions of the plan.

     Otherwise, if $G_{l+l'}$=G_n, then this is the last iteration of the time-bounded construction process, the one that builds a repairing structure that includes G_n. At this point, the construction process finishes since all the actions in the initial plan Π are covered in some repairing structure.

Notice that the more actions in the plan window associated to $\mathcal{T}$, the longer the time t_s the Reactive Planner will have to create the next repairing structure and, in principle, the longer the plan window of the new search space. Moreover, if an iteration does not use up all the time to build $\mathcal{T}$, the remaining time is added to the next t_s, thus giving more time for the construction of the next repairing structure.

Another important remark is that during the construction of $\mathcal{T}_1$, the plan execution is idle until $\mathcal{T}_1$ is built whereas for the subsequent structures, the time-bounded process is building $\mathcal{T}_i$ while the actions of the plan window of $\mathcal{T}_{i-1}$ are being executed. This iterative working scheme gives our model an *anytime* behaviour, thus guaranteeing the Reactive Planner can be interrupted at anytime and it will always have a repairing structure available to attend an immediate plan failure.

3.4.3.1 Estimating the size of the repairing structure

Our main objective is to guarantee that $\mathcal{T}$ is generated within t_s so that $\mathcal{T}$ is always available when a failure arises. That is, we want to ensure that the search space for the upcoming actions is available when the Execution module starts the execution of such actions. If we can ensure that some $\mathcal{T}$ always exists when an action is being executed, then the only operation that needs to be done to repair an action failure is to find a recovery plan in $\mathcal{T}$.

The time limit t_s to build some $\mathcal{T}$ is determined by the time of an execution cycle and the number of execution cycles (actions) in the plan window of the preceding repairing structure. The cycle time depends on whether we are dealing with a simulated or real system and the characteristics of the domain. Some domains may require a relatively large execution cycle time (e.g., 10 sec.), but others may be much shorter (e.g., 10 ms.). In our experiments of Chapter 4, we will assume an execution cycle time of 1000 ms.

In order to guarantee that $\mathcal{T}$ is created within t_s, we estimate the time to generate $\mathcal{T}$ with respect to some selected l and m through an estimating function $\delta(l, m)$ that predicts the value of the real time to generate $\mathcal{T}$ by using a regression model. The regression model is calculated off-line because a training and testing stages are required to fit the model (see Chapter 4).

$$\{l, m\} = \underset{\substack{l \in [2, x], m \in [l+1, y] \\ \delta(l,m) < t_s}}{\arg\max} \delta(l, m) \tag{3.4.3}$$

The Reactive Planner uses Equation 3.4.3 to find the values of l and m for which $\delta(l, m)$ attains its largest value within the time limit t_s. Through this maximization process we compute for every pair of values of l and m whether or not the value of $\delta(l, m)$ is smaller than t_s. The initial values are $l = 2$ and $m = l + 1 = 3$ because we want to build a repairing structure that comprises at least two actions of the original plan ($l = 2$) and at least one action to repair the first action of the plan window is required ($m = 3$). The value of m is progressively increased by 1 until $\delta(l, m) > t_s$. Then, the value of l is increased by 1 and m resets to $l + 1$. The upper bounds of the parameters l and m (x and y, respectively, in Equation 3.4.3) are conditioned accordingly to the value of t_s. More specifically, the overall process is as follows:

1. Initially, set $l = 2$ and $m = 3$.
2. Progressively, increase the value of m by 1 while $\delta(l, m) < t_s$.
3. When $\delta(l, m) > t_s$, check if:
   (a) l equals the number of actions of the plan (in this case, the estimating process has already covered the entire plan), or
   (b) m is equal to $l + 1$, in which case no longer value of l will fulfill $\delta(l, m) < t_s$.

   in any of these two cases, the process jumps to step 6.
4. Otherwise, increase l by 1 and reset m to $l + 1$.
5. If $\delta(l, m) < t_s$, go to 2;
6. Return the combination l and m such that $\delta(l, m)$ attains its largest value and $\delta(l, m) < t_s$.

Table 3.1 shows a trace of the estimation process to calculate the first repairing structure $\mathcal{T}_1$ of our Mars scenario in Figure 5.5. We assume the values of $\delta(l, m)$ are estimated by some regression model and that the default value of t_s is 1000 ms. The Reactive Planner performs the following steps:

1. $l = 2$ and $m = 3$; the value of m is progressively increased by 1 while $\delta(l, m) < 1000$ (see $\delta(2, 7)$ in Table 3.1).

Table 3.1: Trace to calculate l and m values with $t_s = 1000$ ms.

l, m	$2,3$	$2,4$	$2,5$	$2,6$	$2,7$	$3,4$	$3,5$	$4,5$
$\delta(l, m)$	255	637	824	960	~~1052~~	795	~~1230~~	~~1084~~
selected				✓				

2. The value of l is increased by 1 ($l = 3$) and m resets to $l+1 = 4$; we repeat the process of increasing progressively the value of m by 1 until $\delta(l, m) > 1000$, i.e. $\delta(3, 5) > 1000$.

3. Then, the value of l is updated to 4 and m resets to $l + 1 = 5$. In our scenario, the Reactive Planner stops in $\delta(4, 5)$ because $\delta(4, 5) > 1000$ and since m is equal to $l + 1$ no longer value of l will fulfill $\delta(l, m) < 1000$. The Reactive Planner returns the combination l and m that maximize the function δ.

In Table 3.1 the upper bounds of l and m are 4 and 7, respectively. The Reactive Planner selects the combination of values which does not exceed the value of t_s and maximizes the function δ (i.e., $\delta(2, 6)$ in Table 3.1). As a final remark, if the first combination of values of l and m exceeds t_s, that is, if $\delta(2, 3) > t_s$, then $(2, 3)$ would be the selected combination as there are no other possible choices.

In summary, the function $\delta(l, m)$ estimates for some fixed values of l and m, the time of generating $\mathcal{T}$; then, Equation 3.4.3 checks whether or not $\delta(l, m)$ exceeds t_s, and returns the values of l and m for which the value of $\delta(l, m)$ attains its maximum value without exceeding the value of t_s.

3.5 Conclusions

In this chapter, we have presented a single-agent architecture that comprises a general reactive planning and execution model that endows an execution agent with monitoring and execution capabilities. The RPE model integrates a Reactive Planner module that provides the execution agent with fast responsiveness to recover from plan failures in planning applications. Thus, the mission of an execution agent is to monitor, execute and repair a plan, if a failure occurs.

The key element of the Reactive Planner is the repairing structure, a search space composed of partial states that encodes recovery plans for potential failures that may occur in a portion of a plan. Repairing structures are created with an iterative time-bounded construction that guarantees the Reactive Planner will always have a repairing structure available to attend an immediate plan failure. Once the repairing structure is available for some action, the only operation that needs to be done is to search over the partial states of the structure until a recovery plan is found in the repairing structure.

The Reactive Planner contributes with several novelties: (a) it is a domain-independent planner that can be exploited in any application context; (b) it works with the same PDDL encoding of the Deliberative Planner, thus avoiding translation of data; (c) it avoids dealing with unnecessary information of the world, handling specifically the information relevant to the failure; and (d) it performs a time-bounded process that permits to continuously operate on the plan to repair problems during execution.

In the next chapter, we will explain the process to come up with a predictive model and we will show an experimental evaluation of the Reactive Planner module.

Chapter 4

Evaluation of the Reactive Planner

"In evaluating ourselves, we tend to be long on our weaknesses and short on our strengths."

(Craig D. Lounsbrough)

In this chapter, we present the experimental evaluation of the domain-independent Reactive Planner module, which is the main contribution of the previous chapter. More specifically, we conducted two principal experiments:

1. **Selection of the regression model.** In this experiment, we compared two regression models to estimate the time of generating a repairing structure and we selected the one with the smallest estimation error. The selection of the regression model is a relevant issue for the subsequent evaluation of the Reactive Planner since the planner will use the chosen regression model to estimate the size of a repairing structure such as it was explained in Section 3.4.3.1.

2. **Evaluation of the Reactive Planner.** We evaluated the performance and reactiveness of the Reactive Planner. Specifically, we conducted two experiments:

   (a) **Time-bounded construction of repairing structures.** The aim of this experiment is to check if the Reactive Planner is able to build the repairing structure within the available time. We generated the first three structures of the solution plans of various planning tasks. Concretely, to generate one repairing structure, the Reactive Planner uses the selected regression model to find the values of l and m that maximize the function $\delta(l, m)$. Once the values of l and m are known, the Reactive Planner generates the repairing structure and we analyze whether the time taken for building the repairing structure is within the time limit.

(b) **Recovery plan.** We performed several tests to evaluate the performance of the iterative search plan algorithm with the first repairing structure of the solution plans. We also compared our repair mechanism with two other deliberative methods, including an *adapting repair method* and a *replanning mechanism*.

The Reactive Planner module was implemented in Java and all the experiments were run on a GNU/Linux Debian computer with an Intel 7 Core i7-3770 CPU @ 3.40GHz x 8, and 8 GB RAM.

4.1 Selecting the regression model

Our interest lies in comparing a linear regression model against a tree regression model which, in principle, can approximate nonlinear functions, and analyze the error of both approaches when estimating the time of generating a repairing structure. The process of selecting the regression model is divided into four stages:

1. **Generation of the data samples:** We obtained the solution plans for several planning tasks from diverse planning benchmarks[1], and generated random repairing structures for these plans. For each repairing structure, we stored the

[1]Part of the benchmark can be seen at http://planinteraction.cguz.org/resources/.

generation time along with other features which altogether form the representation space of the **data samples** (see Section 4.1.1).

2. **Learning the linear regression model:** Given as input the data samples, we employ cross-validation to fit the linear model that estimates the time of generating a repairing structure.

3. **Learning the bagging model:** Given as input the data samples, we trained the bagging model that estimates the actual time needed to build a repairing structure.

4. **Testing the two regression models:** Given as input the data samples, we tested the two learning models and analyzed the estimated time versus the real time needed to generate the repairing structures. We then compared the results of the two models and we selected the one that better approximates the actual time of generating a repairing structure and returns the minimal prediction error.

4.1.1 Data samples

We selected four planning domains along with their corresponding problems or **planning tasks** (nine planning tasks for each domain) from the benchmarks collections used in the *International Planning Competition* (IPC) [2]:

- the `rovers` domain is a space exploration scenario from the IPC of 2002 (the domain is shown in Appendix B.1).
- the `logistics` domain is a transport scenario from the IPC of 2000 (the domain can be seen in Appendix E.1).
- the `driverlog` domain is a vehicle routing scenario from the IPC of 2002.
- the `parcprinter` domain is a manufacturing scenario from the IPC of 2008.

[2]http://ipc.icaps-conference.org/

These four planning domains feature different characteristics, structural properties and topology of the search space in order to make our Reactive Planner as domain-independent as possible. We solved 36 planning tasks, nine problems of each domain, with LAMA planner [90]. For each planning task $\mathcal{P} = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, we obtained a solution plan Π and for each Π we generated repairing structures with random values of l and m, generating a total of 6,788 repairing structures.

Our aim is to use the collection of 6,788 repairing structures to learn the parameters that better explain the size of a repairing structure $\mathcal{T}_i$ associated to a plan window $[a_1, \ldots, a_l]$. Ultimately, our objective is to estimate the time of generating $\mathcal{T}_i$, what can be done by estimating the size of $\mathcal{T}_i$. Predicting the development of a search tree based on the data of the plan window and the only available node of the search tree, the root node, is not an easy task. We selected the components or features of the repairing structures that directly or indirectly affect the generation of the search tree and let the learning model choose the weights of the variables that better explain the tree generation.

As we detailed in Section 3.4.3.1, the Reactive Planner calculates the size of the repairing structure that can be created within the time limit by trying different values of the plan window length l and maximum depth tree m. Thereby, we could simply extract l and m out of the 6,788 repairing structures collection. However, the plan window l is an indirect and not very informative parameter to estimate the actual size of the search tree, reason why we selected other features of the plan window that will provide much more helpful information for the estimation.

In the following, we analyze the features that can be extracted from the 6,788 repairing structures to make up the sample dataset. Given $\mathcal{T}$, a repairing structure of the 6,788-item collection, we distinguish, on the one hand, the features that provide indication of the time or size of generating $\mathcal{T}$, which ultimately is the variable that the Reactive Planner needs to estimate to ensure a time-bounded construction of a repairing structure. On the other hand, we identify another set of features related to the plan window associated to $\mathcal{T}$. Overall, we extracted 9 parameters (features) from each repairing structure $\mathcal{T}$ (observation), that we classify into two groups:

Group 1: parameters related to $\mathcal{T}$.

1. *real time to generate* $\mathcal{T}$ (t): this is ultimately the variable to predict.
2. *number of nodes of* $\mathcal{T}$ (N): N can be used to derive the time of generating $\mathcal{T}$ if we are able to figure out the average time of expanding a node of the tree.
3. *branching factor of* $\mathcal{T}$ (b): b can not be directly obtained from $\mathcal{T}$ but can be calculated, thus providing an indirect measure of the size of $\mathcal{T}$.
4. *depth of* $\mathcal{T}$ (m): this provides the deepest level of a leaf node.

Group 2: parameters related to the plan window.

5. *number of fluents in the root node* G_l *of* $\mathcal{T}$. This parameter is directly related to the size of the state space of $\mathcal{T}$. As it was explained in Section 3.4.1, the generation process expands G_l considering the actions that affect the fluents in G_l. That is, the more fluents in G_l, the more actions to expand G_l. For instance, the number of fluents of the root node G_2 of $\mathcal{T}_1$ in Figure 3.7 is five (see the fluents of G_2 in Figure 3.6).
6. *number of relevant variables in the plan window of* $\mathcal{T}$. The number of relevant variables is an important parameter because it helps reduce the complexity of $\mathcal{T}$ during the generation process. As we explained in Section 3.4.1.2, we only consider the relevant variables involved in the preconditions and effects of the actions of the plan window $[a_1, \ldots, a_l]$ because these are the only fluents that will actually need to be repaired. In the repairing structure $\mathcal{T}_1$ of Figure 3.7 associated to the plan window $[a_1, a_2]$ of the plan Π of Figure 5.5, the number of relevant variables associated to $\mathcal{T}_1$ is five; i.e., all the variables involved

in the set $pre(a_1) \cup eff(a_1) \cup pre(a_2) \cup eff(a_2)$ = {`loc-B`, `link-w`$_2$`-w`$_1$, `locs-s`$_1$, `analyze-B`, `have-B`} (see the preconditions and effects of the actions in Appendix A.1.1).

7. *number of relevant variables which also appear in the fluents $\langle v, d' \rangle$ of the root node G_l of $\mathcal{T}$*. In other words, the number of relevant variables of parameter 6 that are included in the fluents of the parameter 5. The reason of considering this parameter 7 is that, as we explained in Section 3.4.1, the expansion of the root node G_l only considers the actions $\mathcal{A}$ of the planning task that modify a fluent $\langle v, d' \rangle$ of G_l and whose v is a relevant variable. In the example of Figure 3.7, the value of the parameter 7 is two because the only relevant variables that are included in the fluents of G_2 are `loc-B` and `have-B`.

8. *sum of the cardinality of the domains of the relevant variables of the plan window (parameter 6)*. In the generation process, a node G is expanded by considering all the actions that affect a relevant variable v. Thus, the more values in the domain of v, the higher the likelihood of using more actions throughout the expansion of $\mathcal{T}$ and, thereby, the more complexity of $\mathcal{T}$. Following with the same example, the sum of the cardinality of the domains of the relevant variables of the parameter 6 is 15. Specifically, the domain of the relevant variables of the parameter 6 along with their cardinalities are:

   - $\mathcal{D}_{\texttt{loc-B}} = \{\texttt{w}_1, \texttt{w}_2, \texttt{w}_3\}$ with a cardinality of three.
   - $\mathcal{D}_{\texttt{link-w}_2\texttt{-w}_1} = \{\texttt{true}, \texttt{false}\}$ with a cardinality of two.
   - $\mathcal{D}_{\texttt{locs-s}_1} = \{\texttt{w}_1, \texttt{w}_2, \texttt{w}_3, \texttt{NONE}\}$ with a cardinality of four.
   - $\mathcal{D}_{\texttt{analyze-B}} = \{\texttt{true}, \texttt{false}\}$ with a cardinality of two.
   - $\mathcal{D}_{\texttt{have-B}} = \{\texttt{s}_1, \texttt{s}_2, \texttt{r}, \texttt{NONE}\}$ with a cardinality of four.

9. *number of actions that affect a relevant variable of the plan window*. In order to reduce the complexity of $\mathcal{T}$, only the actions that modify the relevant

variables in the parameter 6 are considered during the tree expansion (see Section 3.4.1.2). For example, the number of actions of $\mathcal{A}$ that changes the value of a variable of the parameter 6 is 21.

The parameters of group 2 have also an impact on the size of a repairing structure. Particularly, parameters 5 (number of fluents in the root node of $\mathcal{T}$) and 7 (number of relevant variables which also appear in the fluents of the root node of $\mathcal{T}$) are directly related to the root node and, in principle, they have a greater impact in the generation of the tree, in contrast to parameters 6 (number of relevant variables in the plan window of $\mathcal{T}$), 8 (sum of the cardinality of the domains of the relevant variables of the plan window) and 9 (number of actions that affect a relevant vairable of the plan window) that are more related to the size, structure and composition of the plan window. As we commented before, we do not include the plan window length l as a parameter of the data samples because it would yield poorly estimated models. On the contrary, parameters 5 to 9 are much more informative because they affect directly or indirectly the items included in the length of the plan window. All in all, the estimation of the Reactive Planner depends on the length of the plan window and depth of the tree. We preserve the value of $m > l$ in our estimation model to ensure that a repairing structure will comprise at least one action to repair the first action of the plan window.

4.1.2 Learning the linear regression model

Multiple linear regression [22] is a statistical technique that predicts or explains the value of a dependent variable (y) by considering more than one explanatory features (x_i). Hagerty demonstrated that linear regression produces less accurate predictions when y is a variable that presents long range values [56]. This is precisely the case of the variable t, the time of generating a repairing structure, because time is expressed in milliseconds and may contain outliers. Similarly, the number of nodes N of a repairing structure is a variable that can take on values in a long range.

The columns 1 and 2 of Table 4.1 presents the minimum and maximum values of t and N, respectively, of the repairing structures of the sample dataset. Consequently, estimating the time or number of nodes of a repairing structure $\mathcal{T}$ with a linear regression model is not feasible because it may produce highly inaccurate predictions.

Table 4.1: Range of values of time t, number of nodes N and branching factor b.

description	t **(ms)**	N	b
Minimum value	1	13	1
Maximum value	4,507,382	242,495	33

For the aforementioned reasons, we opted for selecting the branching factor (b) as the variable to estimate with the linear regression model since the values of b are restricted to a much smaller range, as can be seen in the column 3 of Table 4.1. Obviously, b is not a direct measure of the time or size of a repairing structure but, by using appropriate formulas and inferences, we can obtain the number of nodes N out of b and we can then infer the time t out of N. The parameters 4 to 9 that were presented in Section 4.1.1 will be the explanatory variables x_i, that is, the variables that will be used to estimate b. Specifically, we name the variables as follows:

- x_1: parameter 4 (depth m of $\mathcal{T}$).
- x_2: parameter 5 (number of fluents in G_l of $\mathcal{T}$).
- x_3: parameter 6 (number of relevant variables in the plan window of $\mathcal{T}$).
- x_4: parameter 7 (number of relevant variables which appear in the fluents $\langle v, d' \rangle$ of G_l of $\mathcal{T}$).
- x_5: parameter 8 (sum of the cardinality of the domains of the relevant variables of the plan window).
- x_6: parameter 9 (number of actions that affect a relevant variable in the plan window).

Once the values of the six parameters of the 6,788 repairing structures are extracted, we calculate their branching factor b and complete the sample dataset. We recall that b is a non-observable value of the repairing structures so it must be calculated. In general, there exist many factors that determine not only the size of the tree but also its form. Therefore, it is safe to say that the search trees are irregularly shaped, what makes it difficult to calculate the branching factor. We based our calculations on the effective branching factor [81][3] because this measure is reasonably independent of the maximum depth of the tree and usually it is fairly constant for sufficiently hard planning tasks. Although the effective branching factor b cannot be written explicitly as a function of the depth m and the number of nodes N, we can design a plot of b versus N for some value of m (see Appendix F.1) and come up with the following formula:

$$N = \quad (b + 0.44)^m \tag{4.1.1}$$

By using formula 4.1.1, we calculate the branching factors of the 6,788 repairing structures to complete the sample dataset. Finally, the samples that will be used to learn the linear regression model will have the following features:

- b, the dependent variable y.
- variables x_1 to x_6, the explanatory features.

4.1.2.1 Selecting explanatory parameters

The first step for learning the linear model is to select from the input data samples the parameters x_i that better explain the behaviour of the model to estimate the variable $y = b$, the branching factor of the repairing structures. The objective is to determine the relationships between the input features x_i of the data samples, and

[3]The effective branching factor is the number of children generated in each node if the repairing structure were a uniform tree.

their roles, either alone or in conjunction with others, in describing the response of the branching factor b.

In the literature, various methods have been proposed for selecting some subsets of parameters, which consist in either adding or deleting one or more parameters at a time according to a specific criterion [31]. These procedures are generally referred as stepwise methods, which consist in two basic ideas called *adding selection method* (adding one or more variables in each step) and *elimination method* (removing one or more variables in each step). In our case, we opted for eliminating a set of parameters at a time from all the candidate parameters. In addition, we show that the deletion of parameters does not considerably worsen the model with respect to the full model which includes all the explanatory variables.

$$\hat{b} = \quad w_0 + \sum_{i=1}^{6} w_i * x_i \tag{4.1.2}$$

R^2:	0.704
adjusted R^2:	0.701

Table 4.2: Correlation matrix between dependent variable b and parameters

parameters	b
x_1	-0.463
x_2	-0.409
x_3	0.092
x_4	0.144
x_5	0.081
x_6	0.468

Initially, we calculated the linear model with all candidate parameters x_i (see Equation 4.1.2) and we analysed the correlation matrix between each parameter and the real values of b (see Table 4.2). We see some interesting relationships. It appears that high values of x_6, the number of relevant actions that modify the relevant variables of the plan window, increase the branching factor (positively related), and high values of x_1 and x_2, the depth m and number of fluents in the root node, respectively, are strongly related in decreasing values of b (negatively related). The least significant parameters in the correlation matrix are x_3 and x_5. The parameter x_4 is also loosely related to b in comparison to x_1, x_2 and x_6. In view of these results, we remove the parameters with the least correlation with b; i.e. x_3, x_4 and x_5.

Some studies consider that parameters with correlation coefficients below 0.5 are not significant to the linear model. In our case, we can observe that none of the coefficients reach such value, what may be an indication that the linear model will not properly explain the behaviour of the branching factor. Nevertheless, we decided to design the linear model with the best three explanatory variables and then compare it with a non-linear model (see Section 4.1.4).

$$\hat{b} \simeq f(x) \leftarrow w_0 + w_1 x_1 + w_2 x_2 + w_6 x_6 \tag{4.1.3}$$

R^2:	0.689
adjusted R^2:	0.687

In order to confirm that x_3, x_4 and x_5 are not significant parameters for the linear regression model, we used the coefficients of determination R-squared (R^2) and adjusted R^2, which are statistical measures of how close the data are to the fitted model. The R^2 is always between 0 and 1, where 0 indicates that the linear model does not fit the data at all and 1 indicates that the model perfectly fits the real values of the dependent variable y. In general, the higher the R^2, the better the model fits the data. The adjusted R^2 is a variation of R^2, which is most commonly-cited statistic when we are using multiple explanatory parameters because, in contrast to R^2, it accounts for the number of parameters in the equation. The value of the adjusted R^2 is usually lower than the result for R^2. As we can see in the two Equations 4.1.2 and 4.1.3, removing x_3, x_4 and x_5 slightly affect the R^2 (this value will always decrease when a parameter is removed) and the adjusted R^2, confirming our hypothesis that x_3, x_4 and x_5 are not significant parameters for the linear regression model. Hence, the explanatory variables that we will consider in the linear regression model are x_1, x_2 and x_6. In addition, We calculated the coefficients of determination with a linear model using x_1, x_2 and x_4, i.e., selecting the next best explanatory variable after x_6, and we obtained a value of R^2 equal to 0.456 and adjusted R^2 equal to 0.455, thus confirming that the appropriateness of the selection of x_1, x_2 and x_6 for the design of the linear model.

4.1.2.2 Estimating the time to generate the repairing structure

Once we have the linear regression model, we list the remaining operations to obtain the estimated time of generating a repairing structure:

Linear method:

1. Calculate $\hat{b}$ with the fitted linear regression model.
2. Apply formula 4.1.1 to calculate the total number of nodes $\hat{N}$.
3. Apply formula 4.1.4 to estimate the time $\hat{t}_{\mathcal{T}}$ of generating a repairing structure $\mathcal{T}$. The value of $\bar{t}_regress$ is computed as the average time to apply the regressed transition function $regress$ to a node G' in the repairing structures of the dataset.

$$\hat{t}_{\mathcal{T}} = \quad \hat{N} * t_r e\bar{g}ress \tag{4.1.4}$$

4.1.3 Learning the bagging model

Bootstrap aggregating or Bagging is an *ensemble method* that employs supervised machine learning methods for classification and regression [15]. Bagging trains multiple regression predictors in the ensemble using a randomly drawn subset of the training set. The predictors are combined by averaging the results of predicting a numerical outcome generating an aggregated prediction of multiple models which is less noisy than one calculated by an individual model.

Although there are several models (boosting, stacking, random forest and others), we decided to use the bagging model because as a tree ensemble model it offers better predictions and it is a more stable model than other regression models [43]. Moreover, we emphasize the following reasons:

1. **approximate to nonlinear function problems:** bagging works well with problems that are linearly and non-linearly separable. In general, this is an advantage for tree ensemble models.

2. **performs well with large datasets:** it allows analysing significant amounts of data using standard computing resources in reasonable time.

3. **white box model:** tree ensemble models work as a white box; that is, if a condition is observable in a model, the explanation for the situation is easily explained by boolean logic (model tree). By contrast, in a black box model, the explanation for the predictions is typically difficult to understand, as for instance in an artificial neural network model.

4. **over fitting:** bagging takes care of over fitting in contrast to other ensemble models like Boosting that even though shows better predictive accuracy than bagging, it tends to over-fit the training data as well.

In order to understand the bagging model, let's consider first the regression problem. Suppose we fit a model to the training sample set $D=\{d_1, d_2, \ldots, d_n\}$, obtaining the prediction $\hat{f}(x)$ at any given testing input x. As shown in Figure 4.1, bagging obtains the prediction as the average of the predictions of a collection of predictors with **bootstrap samples** [4], thereby reducing the variance of the prediction. For each bootstrap sample (new training sets) D_c, $c=\{1, 2, \ldots, C\}$, bagging

[4]The basic idea of bootstrap sample is to randomly draw datasets with replacement from the training data, each sample the same size as the original training set. This is done C times, producing C bootstrap datasets.

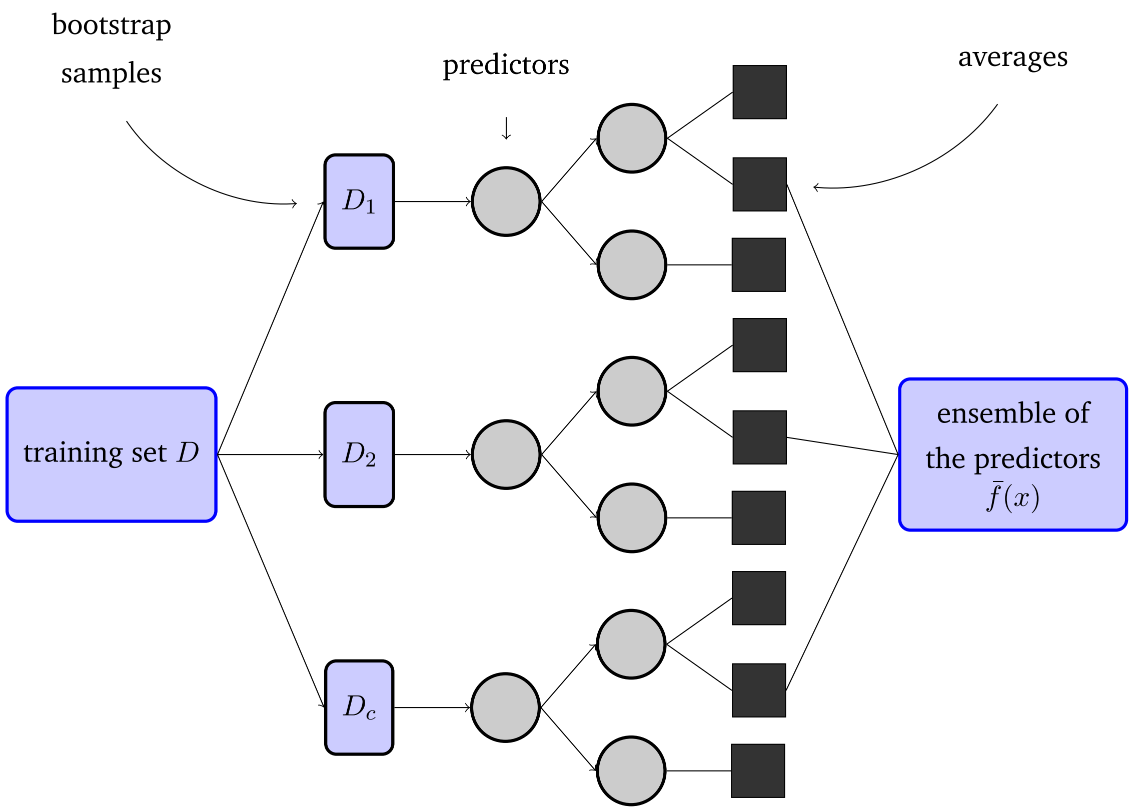

Figure 4.1: Operation of the bagging model, using model tress as base models.

fits a predictor $\hat{f}_c(x)$. Then, bagging estimates the final prediction as a uniformly weighted average defined by:

$$\bar{f}(x) = \frac{1}{C} * \sum_{c=1}^{C} \hat{f}_c(x) \tag{4.1.5}$$

More specifically, bagging manipulates the training dataset to generate multiple explanatory parameters by sampling with replacement. In contrast to the linear regression model, bagging can estimate variables with a long range of values, like the time to generate the repairing structure. Hence, we opted for estimating directly the time of generating a repairing structure instead of the branching factor or number of nodes.

4.1.4 Comparing the two regression models

In this section, we evaluated and compared the two regression models in order to select the one with the best accuracy to estimate the real time of generating repairing structures. We used k-fold cross validation to test how well the two approaches (the linear method and bagging model) are able to get trained by some data and then predict data they have not seen. Other techniques, however, train the model using all the sample dataset and evaluate the fitted model with the same dataset (data that the model has seen before) leading to an overfitted model. Note that we wish to compare the predicted time of both approaches with respect to the real time. Since the linear model only predicts the branching factor, we actually calculated the estimated value of the time for each repairing structure with the so-called linear method exposed in Section 4.1.2.2, which involves calculating the branching factor, estimating the number of nodes and then estimating the time. Thereby, we compared the final result of estimating the time with the linear method (which uses the linear model) and the bagging model.

4.1.4.1 K-fold cross-validation

The purpose of the k-fold cross-validation is to compare the two approaches. We applied the 10-fold cross validation by training the linear and bagging models 10 times on 90% of the data and ensuring that each data point ends up in the 10% test set exactly once. We have therefore used every data point to contribute to an understanding of how well the two models perform the task of learning from some data and predicting some new data. More concretely, our sample dataset is partitioned into $k = 10$ equal sized subsamples of 678 samples. A single subsample is retained as the validation data for testing the model, and the remaining $k - 1$ subsamples are used as training data. The cross-validation process is then repeated k times (the folds) using each of the k subsamples exactly once as the validation data. Hence, we trained the two models k times and obtained k estimations of the out-of-sample error.

In summary, we checked the two models with a systematic approach, in which every instance is used for model testing exactly once giving us some amount of confidence that the error is not too noisy. Next, we will see which model proves better at predicting the test set points.

Additionally, since the linear method procedure to estimate the time is very different from the bagging model, which directly estimates the time, we were also interested in applying the same procedure to a branching factor estimated with the bagging model. Hence, we tested three approaches:

1. the linear method as explained in 4.1.2.2.
2. the bagging method, which computes the estimated branching factor and then applies the same procedure to estimate the time as the linear method.
3. the bagging model.

Table 4.3: Errors obtained with the three approaches using 10-fold cross validation.

measure of error [5]	**linear method**	**bagging method**	**bagging model**
Root Mean Square Error (RMSE):	135,310 ms	126,768 ms	18,445 ms
Mean Absolute Error (MAE):	14,038 ms	11,256 ms	2,147 ms
Mean Absolute Percentage Error (MAPE):	1,403%	1,125%	214%

Table 4.3 describes the results of applying the 10-fold cross-validation to the three approaches. We show three measures of error. The RMSE represents the standard deviation of the residuals; i.e., the difference between the real values and the estimated values of the time of generating a repairing structure; and the MAE is the average of the absolute residuals. RMSE and MAE are also expressed in milliseconds; a good threshold is defined as a value that is closer to the **minimum**

[5]See Appendix F.2 for formulas and a small description of each measure of error.

value of the estimated variable and a bad threshold is a value that is closer to the **maximum** value (see Table 4.1 which shows the minimum and maximum values of t out of all the collected repairing structures). As we can see, the bagging method is better than the linear method, but the estimated values are too high in relation to the minimum value of t. On the contrary, the bagging model presents better results than the linear and bagging methods and its estimation is closer to the real time values, as it is shown in the RMSE and MAE.

The MAPE is calculated as the average of the absolute value of the residuals divided by the real time. The MAPE measures the size of the error in percentage terms. When the predicted variable is within a small range of values, the upper limit of MAPE is between 0% and 100%, but when the range of values is too large, it is not possible to define an upper limit. Again, we can see that the bagging model has better results than the linear and bagging methods. Although MAPE also takes a fairly high value (214%) in the bagging model due to the large range of values of the time variable, this value is significantly lower than the value obtained with the linear method (1,403%) and bagging method (1,125%).

We designed an additional experiment in which we fitted the three approaches with the complete dataset. Table 4.4 shows the results of the three approaches for a dataset of 32 new repairing structures obtained from planning tasks of the `rovers`, `logistics`, `driverlog` and `parcprinter` domains. More specifically, for each planning task, we estimated the real time of generating the first repairing structure with different values of l and m, and we calculated the parameters of Group 2 related to the plan window (see Section 4.1.1).

Table 4.4: Real time and estimated time of the three approaches for generating a repairing structure.

	task	l	m	**time**	**linear method**	**bagging method**	**bagging model**
				$t_{\mathcal{T}}$	$\hat{b} \to \hat{N} \to \hat{t}_{\mathcal{T}}$	$\hat{b} \to \hat{N} \to \hat{t}_{\mathcal{T}}$	$\hat{t}_{\mathcal{T}}$
`rovers` domain	1	2	6	39	532	103	46
	1	5	6	68	680	331	58
	2	2	6	17	566	68	102
	2	5	6	35	553	164	82
	2	4	8	36	466	281	185
	2	5	9	37	512	464	482
	9	2	5	361	740	669	355
	9	4	5	290	882	609	318
	10	2	5	234	713	345	586
	10	4	6	12,661	1,522	8,535	12,134
`logistics` domain	1	2	6	46	892	67	62
	1	5	6	293	1,144	358	286
	2	2	6	43	883	76	46
	2	5	6	131	901	215	144
	2	4	8	1,398	283	214	1,581
	9	2	6	219	179	109	218
	9	4	6	451	213	173	496
`driverlog` domain	1	2	6	65	920	89	72
	1	5	6	503	1,480	22,400	624
	2	2	6	42	1,149	81	36
	2	5	6	39	1,172	214	51
	2	4	8	381	420	771	852
	2	5	9	7,873	859	3,463	8,400
	9	2	6	2,024	3,124	3,201	2,085
	9	4	6	2,578	7,725	3,762	2,825
	10	5	6	18,366	17,277	36,746	18,616
`parcprinter` domain	1	2	6	419	1,525	529	415
	1	5	6	13,178	4,646	8,893	12,697
	2	4	8	111	606	247	136
	2	5	9	1,961	964	931	1,628
	9	2	6	208	1,027	91	216
	9	4	6	2,819	1,128	685	2,932

1. the column **task** denotes the particular planning task for which we estimated the time of the first repairing structure.

2. the columns l and m indicate the plan window length and the maximum depth of the first repairing structure, respectively.

3. the column **time** shows the real time to generate the first repairing structure with the specified values of l and m.

4. the remaining columns show, for each approach, the estimated time to generate the first repairing structure.

As we can see in Table 4.4, the bagging model obtains much better results than the other two approaches since the estimated values are much closer to the real time. In our particular case, the Reactive Planner uses the estimated time to calculate the maximum-size repairing structure that can be built within a given time limit (see Section 3.4.3.1). Let $t_{\mathcal{T}}$ be the real time of generating a repairing structure $\mathcal{T}$, $\hat{t}_{\mathcal{T}}$ the estimated time and t_s the time available to the Reactive Planner to build the repairing structure; following, we expose an analysis of the results of Table 4.4:

1. **overestimated values of the approaches.** If $t_{\mathcal{T}} < \hat{t}_{\mathcal{T}}$, then we can ensure the Reactive Planner will be able to build the search space within the available time because, as we explained in Section 3.4.3.1, the Reactive Planner will always choose a repairing structure whose estimated time is below t_s; therefore, we have $t_{\mathcal{T}} < \hat{t}_{\mathcal{T}} < t_s$. However, if $\hat{t}_{\mathcal{T}}$ largely overestimates $t_{\mathcal{T}}$ ($t_{\mathcal{T}} << \hat{t}_{\mathcal{T}}$), then the Reactive Planner will discard such repairing structure if it happens that $t_s < \hat{t}_{\mathcal{T}}$, when it actually may be the case that the real time of generating the structure is within the deadline; that is, $t_{\mathcal{T}} < t_s < \hat{t}_{\mathcal{T}}$. Thus, the risk of overly large overestimates is that we could rule out choices when there is actually enough time to build the repairing structure within the available time.

For instance, assuming t_s is equal to 500 ms in the `logistics` domain task 2, $l = 2$ and $m = 6$; the linear method will discard this configuration due to $t_s < \hat{t}_{\mathcal{T}}$ (500 < 883), even though the real time is $t_{\mathcal{T}}$=43. However, the bagging model will consider this configuration although it slightly overestimates the real time value. This is because the overestimate of the bagging model is close to the real time and so within the time limit (43 < 46 < 500). The same happens in the `driverlog` domain task 2, $l = 5$, $m = 6$. Thus, we are interested in a model whose estimated values are not overly large and are close to the real time values, such as the estimates of the bagging model.

2. **underestimated values of the approaches.** If $\hat{t}_{\mathcal{T}} < t_{\mathcal{T}}$, then it exists some risk that the Reactive Planner runs out of time iff it holds $\hat{t}_{\mathcal{T}} << t_{\mathcal{T}}$ and $t_s < t_{\mathcal{T}}$. The closer $t_{\mathcal{T}}$ to $\hat{t}_{\mathcal{T}}$, the lower the risk. More specifically, if $\hat{t}_{\mathcal{T}}$ overly underestimates $t_{\mathcal{T}}$ then the Reactive Planner might select a configuration which cannot actually be built within the available time t_s; that is, the Reactive Planner will run out of time before being able to complete the repairing structure.

   For instance, assuming t_s is equal to 1,500 ms in the `parcprinter` domain task 9, $l = 4$, $m = 6$; the linear method will select this configuration due to $\hat{t}_{\mathcal{T}} < t_s$ (1,128 < 1,500), and the Reactive Planner will run out of time because the real time of building the repairing structure is actually much higher than the estimate, $t_{\mathcal{T}}$=2,819. In contrast, the bagging model will discard this configuration because its estimated value (2,932) exceeds the time limit.

   Table 4.5 shows the errors of the underestimates of the three approaches. As we can see, the bagging model only presents a MAPE of 8% compared to the ≈56% of the other two approaches. Besides, the results of RMSE and MAE of the bagging model are better than the results of the linear and bagging methods.

   A closer look at the figures of Table 4.4 show that some underestimated values of the bagging model are a bit far away from the real value (e.g., the parcprinter domain task 1, $l = 5$, $m = 6$) and others are very close to the real time value (e.g., the logistics domain task 9, $l = 2$, $m = 6$ or the rovers domain task 9, $l = 2$, $m = 5$). Moreover, the bagging model looks like to underestimate the values when the real time values are too big (time > 10 secs., e.g., the rovers domain task 10, $l = 4$, $m = 6$ or the parcprinter domain task 1, $l = 5$, $m = 6$).

Table 4.5: Errors for the underestimates of the three approaches.

measure of error	linear method	bagging method	bagging model
Root Mean Square Error (RMSE):	5,295 ms	2,623 ms	221 ms
Mean Absolute Error (MAE):	3,539 ms	1,963 ms	121 ms
Mean Absolute Percentage Error (MAPE):	57%	56%	8%

In summary, the accuracy of the bagging model is confirmed by the results and analysis exposed in this section. For all these reasons, we opted for using the bagging model in our Reactive Planner

4.2 Experimental evaluation

In this section, we present the evaluation of the Reactive Planner module with two planning domains, the rovers and logistics domains. Since we do not use exactly the same domains of the IPC but a modified version (see the rovers domain and logistics domain used in our approach in Appendices B.1 and E.1, respectively), we generated a set of planning tasks for each domain that have a similar complexity to the first ten tasks of the respective IPC benchmarks.

We computed a solution plan for each planning task with the LAMA planner. Given a solution plan Π, we performed two off-line evaluations (with no simulation of the execution of Π) in order to test the performance of the Reactive Planner:

1. **generation of the time-bounded repairing structures for Π:** the purpose of this test is to check if the bagging model of the Reactive Planner is capable of generating the repairing structures for Π within the corresponding time limits.

2. **plan repair:** we simulate various failures in the plans of the `rovers` domain in order to check if the Reactive Planner can find a recovery plan with the repairing structure of the plan window where the failure occurs. In addition, we compared the results of our recovery plan mechanism with two other repair methods [40, 90]. Our primary intention with this experiment is to evaluate the ability of the structures to recover of a plan failure.

4.2.1 Preparing the sample dataset

The generation of the repairing structures is obviously dependent on the particular machine in which we run the experiments. Due to the CPU and RAM capacities of our machine[6], structures with a size higher than $l = 7$ and $m = 11$ take overly long times to be generated, causing sometimes memory overflow. Moreover, this also happens with some structures whose depth is larger than 5 ($m >= 5$). Given these computational limitations, the maximum size of the repairing structures we will be able to generate is $l = 7$ and $m = 11$.

Given that the largest possible repairing structure that the Reactive Planner will handle is $l = 7$ and $m = 11$ and that the largest deadline will be thus 7 seconds, we will consider outliers and discard from the dataset the samples that were generated in more than 10 seconds. We believe that it is acceptable to drop samples that took more than 10 seconds because the Reactive Planner will never have more than 7 seconds to build a repairing structure and, as we observed in Section 4.1.4, the bagging model presents a more pronounced trend to underestimate when the time of generating a repairing structure is overly large.

Note that, on the other hand, we are not particularly interested in obtaining

[6]The CPU is an Intel 7 Core i7-3770 @ 3.40GHz with 8 GB of RAM.

very large trees, not only because their generation may exceed the time limit of the Reactive Planner but also because the larger the tree, the longer the process of finding a recovery plan within the repairing structure. Ultimately, our objective is to provide quick responsiveness to potential plan failures.

4.2.2 Time-bounded repairing structures

In this section, we analyze the accuracy of our Reactive Planner to generate the repairing structures of 12 tasks of the `rovers` domain and 10 tasks of the `logistics` domain. More specifically, we evaluate the timely generation of the repairing structures of the solution plans for the 22 planning tasks.

As we explained in Section 3.4.3, the Reactive Planner generates various repairing structures for a solution plan Π through an iterative time-bounded construction process where, in the first iteration, the Reactive Planner generates $\mathcal{T}_1$ for a plan window of length l within a time limit t_s of one cycle time. The value of the cycle time depends on whether we are dealing with a simulated or real system and the characteristics of the machine. In our experiments, we consider 1000 ms to be a reasonable deadline for the simulation of an action execution, i.e. a monitor/repair/execution cycle, because the monitor takes $\approx$ 96 ms, the repair takes $\approx$ 170 ms (this is analyzed in more detail in Section 4.2.3) and the remaining time is assigned for the execution. On the other hand, the limit of 1000 ms (one cycle time) is only applied to the generation of the first repairing structures. For the construction of the subsequent structures $\mathcal{T}_i$, the available time of the Reactive Planner will depend on the number of actions in the plan window of the preceding repairing structure $\mathcal{T}_{i-1}$. This is so because while the Reactive Planner is executing the actions of one plan window, it is simultaneously calculating the repairing structure of the subsequent plan window. Therefore, the most critical operations are those concerned with the first repairing structure, which are limited to one execution cycle.

Tables 4.6 and 4.7 show the results of generating the structures for the 12 planning tasks of the `rovers` domain and the 10 planning tasks of the `logistics` domain, respectively. The data in those tables are the following:

Table 4.6: Time-bounded repairing structures of `rovers` domain.

data set complexity						$\mathcal{T}_1$ $(t_s$ =1 sec)				$\mathcal{T}_2$					$\mathcal{T}_3$				
task	rovers	locations	goals (soil)	(rock)	(image)	Π (actions)	(l,m) (size)	*time* (sec)	N (nodes)	Π (actions)	(l,m) (size)	t_s (sec)	*time* (sec)	N (nodes)	Π (actions)	(l,m) (size)	t_s (sec)	*time* (sec)	N (nodes)
1	1	3	1	0	1	6	(2,9)	0.04	30	4	(3,11)	2	1.9	2,887	1	(1,7)	3	0.08	578
2	1	3	1	0	1	6	(4,9)	0.05	167	2	(2,11)	4	0.05	229					
3	1	3	1	1	1	9	(4,10)	0.36	1,021	5	(5,11)	4	5.06	7,715					
4	1	4	1	1	1	8	(2,10)	0.9	8,780	6	(6,9)	2	6.27	61,480					
5	1	4	2	1	1	11	(2,10)	0.58	5,758	9	(6,8)	2	1.63	18,109	3	(3,11)	6	3.21	26,527
6	2	4	2	1	1	11	(3,5)	0.67	2,920	8	(4,8)	3	4.09	18,952	4	(3,10)	4	3.17	19,321
7	2	4	3	1	1	14	(3,9)	1.07	9,420	11	(5,8)	3	2.11	22,545	6	(5,11)	5	4.32	20,595
8	2	5	2	2	1	14	(2,7)	0.72	11,259	12	(2,11)	2	1.98	26,266	10	(5,6)	2	1.83	35,637
9	2	6	3	3	1	18	(2,6)	0.91	13,754	16	(4,5)	2	0.5	5,023	12	(3,7)	4	3.2	27,477
10	2	6	3	3	2	21	(2,8)	0.94	13,378	19	(3,5)	2	2.73	24,338	16	(2,7)	3	2.33	17,112
11	4	10	2	4	3	35	(2,7)	0.73	3,177	33	(2,4)	2	0.12	397	31	(4,5)	2	1.26	3,313
12	4	10	3	4	3	39	(2,7)	0.78	3,167	37	(2,5)	2	1.21	401	35	(4,5)	2	1.15	3,046

	$\mathcal{T}_4$					$\mathcal{T}_5$				
task	Π (actions)	(l,m) (size)	t_s (sec)	*time* (sec)	N (nodes)	Π (actions)	(l,m) (size)	t_s (sec)	*time* (sec)	N (nodes)
1										
2										
3										
4										
5										
6	1	(1,5)	3	0.18	158					
7	1	(1,8)	5	0.21	266					
8	5	(5,10)	5	5.39	43,131					
9	9	(3,8)	3	3.05	24,321	6	(5,7)	3	2.81	55,460
10	14	(2,5)	2	0.39	2,151	12	(3,4)	2	0.65	4,380
11	27	(4,8)	4	3.2	3,916	23	(2,8)	4	0.35	460
12	31	(2,9)	4	2.25	1,193	29	(2,6)	2	0.76	690

Table 4.7: Time-bounded repairing structures of `logistics` domain.

data set complexity					$\mathcal{T}_1$ (t_s =1 sec)				$\mathcal{T}_2$					$\mathcal{T}_3$				
task	planes	trucks	locations	goals (packages)	Π (actions)	(l,m) (size)	*time* (sec)	N (nodes)	Π (actions)	(l,m) (size)	t_s (sec)	*time* (sec)	N (nodes)	Π (actions)	(l,m) (size)	t_s (sec)	*time* (sec)	N (nodes)
1	1	2	4	4	20	(3,7)	0.36	184	17	(7,8)	3	1.86	892	10	(6,10)	7	5.70	1,010
2	1	2	4	5	27	(3,7)	0.33	441	24	(6,9)	3	1.46	594	18	(6,10)	6	4.69	950
3	1	2	4	6	25	(2,9)	0.24	248	23	(4,7)	2	1.27	238	19	(7,10)	4	1.69	642
4	1	3	6	7	36	(4,5)	0.29	397	32	(6,10)	4	2.05	153	26	(5,8)	6	3.32	988
5	1	3	6	8	31	(3,6)	0.46	882	28	(2,10)	3	1.62	132	26	(4,7)	2	0.95	407
6	1	3	6	9	36	(2,8)	0.34	158	34	(3,7)	2	1.21	184	31	(7,8)	3	4.05	4,096
7	1	4	8	10	46	(4,5)	0.48	549	42	(6,8)	4	5.23	1,724	36	(3,10)	6	5.06	2,334
8	1	4	8	11	50	(4,5)	0.26	361	46	(6,8)	4	4.05	904	40	(3,10)	6	4.09	1,470
9	1	4	8	12	42	(3,5)	0.17	174	39	(2,8)	3	1.02	198	37	(3,7)	2	1.62	484
10	2	5	10	13	81	(3,4)	0.37	175	78	(3,7)	3	1.29	263	75	(2,7)	3	1.45	364

	$\mathcal{T}_4$					$\mathcal{T}_5$				
task	Π (actions)	(l,m) (size)	t_s (sec)	*time* (sec)	N (nodes)	Π (actions)	(l,m) (size)	t_s (sec)	*time* (sec)	N (nodes)
1	4	(3,10)	6	0.74	54	1	(1,8)	3	2.78	541
2	12	(7,8)	6	4.08	2,315	5	(5,10)	7	5.06	4,011
3	12	(7,8)	7	7.62	5,424	5	(5,8)	7	5.16	6,371
4	21	(5,9)	5	5.05	2,148	16	(6,10)	5	6.34	5,340
5	22	(6,8)	4	3.89	2,491	16	(6,10)	6	6.06	2,595
6	24	(5,10)	7	1.05	42	19	(5,10)	5	5.02	1,712
7	33	(2,10)	3	1.33	32	31	(3,7)	2	1.43	472
8	37	(5,6)	3	1.44	922	32	(6,9)	5	3.24	962
9	34	(3,9)	3	0.79	80	31	(2,10)	3	0.92	80
10	73	(4,5)	2	1.51	386	69	(3,9)	4	1.09	55

1. **data set complexity** shows the size and complexity of the different planning tasks of each domain. The number of resources (rovers, planes and trucks), locations and goals contribute to increase the complexity of each task.

2. The columns under the label $\mathcal{T}_1$ are the data corresponding to the first repairing structure. Specifically:

   - Π: number of actions of the original solution plan.
   - **(**l, m**)**: l is the length or number of actions of the plan window and m the maximum depth of $\mathcal{T}_1$, respectively.
   - ***time***: real time spent in the construction of $\mathcal{T}_1$.
   - N: number of nodes in the search space of $\mathcal{T}_1$.

   We recall that the time limit to build $\mathcal{T}_1$ is always t_s=1 sec reason why t_s does not appear in the data associated to $\mathcal{T}_1$.

3. Subsequent repairing structures until covering the entire plan Π. In some cases only two repairing structures, $\mathcal{T}_1$ and $\mathcal{T}_2$, were generated, other tasks required three, four or even five repairing structures to cover the whole plan. The columns under the label $\mathcal{T}_2$, $\mathcal{T}_3$, $\mathcal{T}_4$ and $\mathcal{T}_5$ denote:

   - Π: number of remaining actions after discarding the actions that will be executed with the preceding repairing structure. For instance, in $\mathcal{T}_1$ of the task 9 of the `rovers` domain (see Table 4.6), the length of its plan window is $l = 2$ and thereby the number of actions of the plan when the construction of $\mathcal{T}_2$ is initiated is Π=$18 - 2$=16. In the case of $\mathcal{T}_2$ of the same task, the length of its plan window is $l = 4$ and consequently the number of actions of the plan when the construction of $\mathcal{T}_3$ starts is Π=$16 - 4$=12.
   - **(**l, m**)**: l is the length or number of actions of the plan window of $\mathcal{T}_i$ and m the maximum depth of $\mathcal{T}_i$, respectively.

- t_s: deadline to create the repairing structure. The value of t_s for building $\mathcal{T}_i$ other than $\mathcal{T}_1$ is set equal to as many seconds as number of actions covered by the previous repairing structures $\mathcal{T}_{i-1}$. For instance, in $\mathcal{T}_2$ of task 3 of the `rovers` domain, t_s=4 because $l = 4$ in $\mathcal{T}_1$. In $\mathcal{T}_4$ of task 8 of the same domain, t_s=5 because $l = 5$ in $\mathcal{T}_3$.
- ***time***: real time spent in the construction of the repairing structure.
- N: number of nodes in the search space of the repairing structure.

All times are measured in seconds. In Tables 4.6 and 4.7, five repairing structures ($\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$, $\mathcal{T}_4$ and $\mathcal{T}_5$) were created for all tasks except for tasks 1 to 8 of the `rovers` domain, where:

- $\mathcal{T}_1$ and $\mathcal{T}_2$ suffice to cover the full plan Π in tasks 2, 3 and 4.
- $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ suffice to cover the full plan Π in tasks 1 and 5.
- $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ and $\mathcal{T}_4$ suffice to cover the full plan Π in tasks 6, 7 and 8.

We highlight two observations about the repairing structures of the rovers domain in Table 4.6. The first remark is that almost every $\mathcal{T}_i$ was generated within the deadline (*time* $< t_s$), except $\mathcal{T}_1$ of task 7; $\mathcal{T}_2$ of tasks 3, 6 and 10; and $\mathcal{T}_4$ of tasks 8 and 9, which slightly exceeded its time limit. Most notably is the case of $\mathcal{T}_2$ of task 4, which generation time exceeds t_s in more than 4 seconds. The reason of this outlier is that the dataset used to train the bagging model does not contain enough structures with such complex sizes. The dataset has 6,530 samples out of which only 46 structures are of size (l,m)=(6,9). We executed an additional test in which we added ten samples of size $l = 6$ and $m = 9$ in the dataset, giving a total of 6,540 samples. Then, we fitted the model and estimated the size of $\mathcal{T}_2$ of task 4 again. In this case, the Reactive Planner returned a structure of size $l = 6$ and $m = 8$, which generation time also exceeded the time limit, but only in 3 seconds.

The second observation is that, for some repairing structures, we can also observe that the values of (l, m) and t_s are the same but the values of *time* and N

are different. For instance, (l,m)=(2,10) and t_s=1 sec for $\mathcal{T}_1$ of tasks 4 and 5 but the values of *time* and N are fairly different because of the increasing complexity of these tasks in terms of the number of locations and goals, which implies a higher branching factor in each task. In general, the branching factor also depends on the *number of relevant variables in the plan window of* $\mathcal{T}_i$ and, hence, a tree like $\mathcal{T}_1$ of task 5 may result in a smaller search space than the one of $\mathcal{T}_1$ of task 4.

In some structures, like $\mathcal{T}_1$ of tasks 1 and 2, the number of nodes N is small, i.e. the search space is fairly small because the newly generated partial states are all repeated nodes after a certain point. Hence, in those cases the entire state space is quickly exhausted, in contrast to other repairing structures like the ones in tasks 6 to 12.

As we explained in Section 3.4.3.1, the Reactive Planner applies a maximization process that computes for every pair of values of l and m whether or not the value of $\hat{t}_{\mathcal{T}}$ is smaller than t_s and selects the values of l and m for which $\hat{t}_{\mathcal{T}}$ attains its largest value without exceeding the value of t_s ($\hat{t}_{\mathcal{T}} < t_s$). This selection criterion can be sometimes problematic, like in $\mathcal{T}_2$ of task 4, where the Reactive Planner selects the structure of size (l,m)=(6,9), which generation time overly exceeds the time limit, even though there are more combinations of values of l and m that also satisfy $\hat{t}_{\mathcal{T}} < t_s$ and in which *time* may not exceed t_s. Our model is sufficiently flexible and allows us to change the selection criterion according to our needs, providing a stronger guarantee that the Reactive Planner will generate the structure within the time limit. For example, we can change the criterion and select the second largest value of $\hat{t}_{\mathcal{T}}$ or the structure with the smallest value of l. Note that selecting the smallest value of l, i.e. $l = 2$, provides more guarantee that the structure will be generated within t_s because it is very likely that the number of nodes of the tree will not be too large.

Table 4.7 shows the results obtained for the `logistics` domain, where the goal is to find optimal routes for several vehicles which have to deliver a number of packages. All of the repairing structures were generated within the time limit, except for

8 repairing structures that slightly exceeds the time limit. Most notably are the cases of $\mathcal{T}_2$ of task 7, $\mathcal{T}_3$ of task 6 and $\mathcal{T}_5$ of task 4 whose generation time exceeds t_s in $\approx$ 1.26 seconds. In general, the search trees generated in this domain are smaller (lower values of N) than those of the `rovers` domain under the same value of t_s. This is because the branching factor in the `logistics` domain is much lower than in the `rovers` domain since trucks are confined to move in a particular city and, hence, loading a package in a truck will only ramify across the trucks defined in the city where the package is located. The same happens with the planes. In contrast, in the `rovers` domain, rovers are equipped with all functionalities and therefore the branching factor and number of nodes N are considerably higher. We also note that some search spaces take the same *time* to be generated but present a significantly different number of nodes, like $\mathcal{T}_2$ of task 7 and $\mathcal{T}_5$ of task 3 (1,724 and 6,371 nodes, respectively). The is because in $\mathcal{T}_2$ of task 7, many of the evaluated nodes are repeated nodes, which are not counted in the parameter N.

From the above experiments, we can draw the following general conclusions:

1. There are some cases where the generation time exceeds the time limit because the Reactive Planner builds rather large repairing structures. The reason is that the dataset used to train the model does not contain sufficient structures with such complex sizes.

2. In our model, when the repairing structure is not generated within the time limit, the Reactive Planner will not have available a complete repairing structure to find a recovery path. If so, the Reactive Planner will return a *not found solution plan* message when requested a recovery plan and the Deliberative planner will be activated. However, the Reactive Planner can actually use the structure, although it is not completely generated, to find a recovery path that reaches the root node G_l of the structure directly.

3. In highly reactive applications, we can opt for changing the selection criterion

of the Reactive Planner and select small repairing structures (e.g., $l = 2$ or $l = 3$) and depth values of m two or three times larger than l and thus provide a stronger guarantee that the Reactive Planner will generate the structure within the time limit. In other words, selecting lower values of l and m can offer more reactivity with the additional cost that the repairing structure presents less possibilities to find a node in the state space that matches the current world state.

All in all, the results corroborate the accuracy of the bagging model and the timely generation of the repairing structures. Overall, out of all of the experiments carried out in the diverse domains, we can say that 85% of the repairing structures are always generated within the time limit. On the other hand, in contrast to most of reactive planners [8], our model is far more flexible since it is capable of dealing with more than one action and multiple failure states with a single repairing structure.

4.2.3 Repairing plan failures

The objective of this section is to test our repair mechanism explained in Section 3.4.2. We compared the average times to recover from a plan failure with our Reactive Planner, a replanning mechanism and a plan-adaptation mechanism. The failures were randomly simulated in the plan window of the $\mathcal{T}_1$ of the planning tasks of the `rovers` domain described in Table 4.6. We generated a plan failure for each action of the plan window associated to $\mathcal{T}_1$. We changed the value of a fluent in the initial state, thus provoking an erroneous execution in the action a_1; the second failure, applied in the state resulting from a successful execution of a_1, affects action a_2; the third failure affects a_3, assuming the two preceding actions, a_1 and a_2, were correctly executed and so on.

4.2.3.1 Structure of the results table

In this section, we explain the structure of the results presented in Table 4.8, which depicts the outcome of the plan failure repair in the `rovers` domain with three recovery approaches:

1. **Reactive Repair:** In this approach we use the Reactive Planner over the first repairing structure $\mathcal{T}_1$.
2. **Adaptation:** This approach applies the LPG-Adapt mechanism [40], which repairs a failure adapting the actual plan to the new current state.
3. **Replanning:** This approach uses the classical planner LAMA [90] as a deliberative planner to obtain a new plan from scratch (replanning) when a failure is detected.

The row **type** in Table 4.8 denotes the simulated plan failures, which are classified in four categories:

A. Failures generated due to an error in the execution of the action (Navigate ?`r` ?$\texttt{w}_\texttt{f}$?$\texttt{w}_\texttt{t}$). An error of type A is because of: 1) the rover ?`r` is not located in ?$\texttt{w}_\texttt{f}$ at the time of executing the action or 2) the path from ?$\texttt{w}_\texttt{f}$ to ?$\texttt{w}_\texttt{t}$ is blocked and the rover cannot traverse it.

B. Failures that prevent the rover from analyzing the results or taking good pictures to complete its plan mission. This type of failure is caused by: 1) the rover loses the sample (rock or soil) by an unexpected event when it is about to analyze it, or 2) the camera loses calibration before taking the picture in a given waypoint position.

C. Failures that are solved with the help of other rovers when a hardware failure disables the device of a rover. Since fixing the damaged hardware is not an eligible option in the `rovers` scenario, the only possible way to repair this failure is with the help of another rover, as long as this is feasible within the repairing structure.

D. Failures that positively affect the plan execution.

Table 4.8 also describes the following information:

- Π: number of actions of the plan Π at the time of executing the recovery plan mechanism.
- Π': total number of actions of the recovery plan Π'. Failures labeled as **root** in Π' denote that the recovery plans were calculated up to the root node of $\mathcal{T}_1$.
- **reused** Π: number of actions of Π that appear in Π'.
- ***time***: real time of the procedure to repair the plan measured in milliseconds.

4.2.3.2 Analysis of the outcome

In this section, we analyze the results of the experiments with the three repair approaches. We will put more emphasis on the solution plans of our Reactive Planner, which is one of the main contributions of this PhD dissertation. As we can see in Table 4.8, the three approaches were able to find a plan except for failure 4 of task 2. In this case, a recovery plan does not exist because the path from waypoint $\mathtt{w}_1$ to $\mathtt{w}_2$ is blocked and there is no other possible way to navigate to $\mathtt{w}_2$.

In our **Reactive Repair** approach, the solutions found to each type of plan failures depicted in the row **type** of Table 4.8 are as follows:

A. The solutions are about rerouting rovers through other paths using the available travel maps.

B. The recovery implies either exploring the area again seeking a new sample of rock or soil (e.g., failure 2 of task 2) and then analyze it, or calibrating the rover's camera again (e.g., failure 2 of task 4).

C. Failures of this type were found in two cases (failure 3 of task 6 and failure 2 of task 12). The Reactive Planner was capable of solving these two failures because $\mathcal{T}_1$ comprised paths involving the second rover. For instance, in failure 3 of task 6, a hardware failure prevents the rover from analyzing the soil in a specific location and our model repairs the failure with the second rover, which explores the area seeking for a sample of soil, analyzes the sample and communicates the results to the lander.

D. Solutions to failures of type D leverage the *positive* failure, which achieves the effects of the next action to execute and, consequently, the Reactive Planner proceeds with the following action in Π.

Table 4.8: Time and recovery plans for repairing tasks with $\mathcal{T}_1$ in the `rovers` domain.

	tasks	1		2				3				4		5		6			7			8		9		10		11		12	
	failure #	1	2	1	2	3	4	1	2	3	4	1	2	1	2	1	2	3	1	2	3	1	2	1	2	1	2	1	2	1	2
	type	D	A	A	B	A	A	D	A	A	D	A	B	A	A	A	A	C	A	B	A	A	A	A	B	A	A	A	A	A	C
	Π	6	5	6	5	4	3	10	9	8	7	9	8	13	12	11	10	9	16	15	14	16	15	22	21	25	24	35	34	39	38
reactive repair (Reactive Planner)	Π'	5	6 (root)	7	6	4 (root)	✗	9	10	9	6 (root)	10	9	15 (root)	13	12	11	11 (root)	18	16	16 (root)	17	16	24 (root)	22	26	25 (root)	37 (root)	35	41	39
	reused Π	5	4	6	5	3	✗	8	9	8	6	9	8	11	12	11	10	8	14	15	14	16	15	21	21	25	23	33	34	39	38
	*time**	1.07	0.58	0.15	1.46	1.13	0.84	0.16	0.13	0.09	1.44	0.27	0.24	5.19	0.17	0.24	0.20	0.96	1.41	0.16	2.26	0.28	0.22	22.13	0.23	0.50	2.63	4.18	0.42	0.69	0.35
adaptation (LPG-Adapt)	Π'	5	9	9	6	4	✗	13	11	9	6	12	12	13	13	13	15	14	18	19	14	19	20	23	25	28	25	40	37	45	41
	reused Π	5	4	6	5	3	✗	10	8	8	4	9	7	10	10	11	10	9	13	14	12	15	14	18	19	23	21	30	27	29	33
	*time**	54	45	58	52	43	51	45	44	43	53	53	52	55	45	48	45	49	56	49	57	43	53	44	56	44	45	51	49	52	50
replanning (LAMA)	Π'	5	6	7	6	4	✗	11	9	9	5	11	11	15	15	12	10	14	17	17	14	18	16	24	24	28	27	36	34	39	42
	reused Π	5	4	5	5	3	✗	6	4	4	5	5	6	9	7	3	1	2	4	10	0	4	4	13	4	15	3	22	25	21	29
	*time**	36	39	54	38	36	32	39	38	41	39	41	41	42	59	51	51	51	51	52	52	59	57	68	69	74	72	140	166	148	149

*time expressed in milliseconds.

In general, we can classify the solution plans found by our repairing approach (row Π' in Table 4.8) as:

1. plans that benefit from positive failures of type D, and so they contain fewer actions than the original plan Π (e.g. failure 1 of tasks 1 and 3).
2. plans that reuse all the actions in Π (e.g. failure 1 of tasks 2 and 6).
3. plans that reuse only some of the actions in Π (e.g., failure 1 of tasks 5 and 11).

With regard the performance outcomes in Table 4.8 and the summary of statistics in Table 4.9, we can highlight the following conclusions:

- **Reused** Π**:** Comparing the values of **reused** Π of our Reactive Planner and LAMA in Table 4.8, we can observe that the Reactive Planner reuses more actions of Π in the new plan Π', particularly in the most complex tasks 6 to 12. For instance, in failure 1 of task 6, both approaches repair the failed plan with 12 actions, but the Reactive Planner reuses the total number of actions of Π (11 actions) compared with the three actions of the replanning approach. This seems reasonable since LAMA does not repair the failed plan but it computes a new plan from scratch. Moreover, the Reactive Planner also outperforms LGP-Adapt, notably in the failures of the tasks from 7 to 12.

  The first column of Table 4.9 shows the mean and standard desviation of the number of actions of the original plan Π that are reused in the recovery plan Π' for all the failures in Table 4.8. As we can see, LAMA is the approach that presents the worst results, reusing only 51% of the actions of the original plan in average. In contrast, our Reactive Planner is the approach that shows the best results with a high percentage (92%) of reused actions in average.

- **Plan quality:** We can see in Table 4.8 that the plan quality or number of actions of Π' is slightly higher with the Reactive Planner than with LAMA in some cases (e.g., failure 1 of task 12 or failure 3 of task 7), and lower in some other cases (e.g., failure 2 of task 12 or failures 1 and 2 of task 10). LAMA is able to find shorter plans in a few cases because it computes a plan for the new situation without being subject to maintain the actions in Π. Nevertheless, the Reactive Planner returns plans of better quality (fewer actions) than LAMA as shown in the column **increase of** Π' in the summary Table 4.9. This column denotes the percentage increase in the number of actions of Π' with respect to the original plan Π. Thus, a value of, say, 2% indicates that Π' is, in average, 2% longer than Π. The column shows the mean value and standard deviation for all the recovery plans in Table 4.8. The Reactive Planner is the approach that shows the best results, where the recovery plans Π' are, in average, 3.76% longer than the original plan. In contrast, LPG-Adapt is the approach that shows the worst results with an average increase of the plan length of 15.89%.

- **Computation time:** According to the *time* results of Table 4.8, the Reactive Planner is much faster than the other two approaches. The most costly result is presented in the failure 1 of task 9, which runtime is 22.13 ms. In this case, the Reactive Planner must explore the entire search space to find a recovery plan up to the root node. Other examples where the Reactive Planner explores the entire search space are presented in tasks 3 and 5, where the recovery plan reaches the root node. Despite this, Table 4.9 shows outstanding runtime results of the Reactive Planner compared to LPG-Adapt and LAMA, which proves the benefit of using a reactive repair to fix plan failures.

  Table 4.10 shows some results of the Reactive Planner when the search space is exahusted to find a recovery plan. We selected different repairing structures from Table 4.6, and we run our recovery mechanism until the search space is exhausted. As we can see, in the case of a search space of 13,378 nodes ($\mathcal{T}_1$ of task 10), the Reactive Planner takes 54.5 ms. LPG-Adapt and LAMA for the same task 10 found a solution plan in $\approx$ 45 ms and $\approx$ 70 ms, respectively (see the row *time* of the adaptation and replanning approaches for the task 10 of Table 4.8). All in all, in relatively big search spaces like $\mathcal{T}_2$ of task 4, the Reactive Planner only takes a small amount of time in exploring the complete search space (we can say the Reactive Planner guarantees a response time of $\approx$ 156 ms in this complex case).

Table 4.9: Summary of statistics for the three approaches.

	reused (%)		increase of Π' (%)		time (ms)	
	μ	σ	μ	σ	μ	σ
Reactive Planner	92	8	3.76	21.56	1.65	4.05
LPG-Adapt	85	15	15.89	30.41	49.47	4.72
LAMA	51	27	6.62	25.03	62.83	36.99

Table 4.10: Time of searching a plan failure in the worst case.

task	$\mathcal{T}_i$	N	worst case time (ms)
5	$\mathcal{T}_1$	5,758	14.2
10	$\mathcal{T}_1$	13,378	54.5
9	$\mathcal{T}_1$	13,754	56
5	$\mathcal{T}_2$	18,109	65.1
6	$\mathcal{T}_2$	18,952	65.9
8	$\mathcal{T}_3$	35,637	101.65
4	$\mathcal{T}_2$	61,480	155.9

The results of the experimental evaluation presented in this section shows that our Reactive Planner throws outstanding results compared to the adaptation approach, LPG-Adapt, and the replanning approach LAMA. This proves the benefit of using the Reactive Planner to repair plan failures in reactive environments besides avoiding the overhead of communicating with a deliberative planner. In conclusion, we can affirm that our model is a robust recovery mechanism for reactive planning that also provides good-quality solutions and performs admirably well in all the measured dimensions.

4.3 Conclusions

In this chapter, we have presented the evaluation of the Reactive Planner. First, we compared two regression models to estimate the time of generating a repairing structure and we selected the bagging model, which presented the smallest estimation error. Next, we evaluated the performance and reactiveness of our single-agent Reactive Planner by conducting two experiments. The purpose of the first experiment was to check whether the Reactive Planner is able to build repairing structures within the available time and the second experiment evaluated the performance of our recovery mechanism compared with two other deliberative methods.

The results in Sections 4.2.2 and 4.2.3 show that the Reactive Planner outperforms other repairing mechanisms. The most relevant limitation of our model is the machine dependency of the regression model explained in section 3.4.3.1. In order to reproduce the experiments, or to export them to other systems, the training of the regression model must be repeated to adjust the values for a particular processor. This is a common limitation for any machine learning regression model.

In summary, the overall show that there is a 90% likelihood to obtain a repairing structure within the time limit. Additionally, the exhaustive experimentation carried out on the repairing tasks confirms that the repairing structure together with the reactive recovery process is a very suitable mechanism to fix failures that represent slight deviations from the main course of plan actions. The results support several conclusions: the accuracy of the model to generate repairing structures in time, the usefulness of a single repairing structure to repair more than one action in a plan fragment while reusing the original plan as much as possible, and the reliability and performance of our recovery search procedure in comparison with other well-known classical planning mechanisms.

Chapter 5

Multi-Agent Reactive Plan Repair

"Collaboration begins with mutual understanding and respect."

(Astronaut Ron Garan)

In the previous two chapters, we have presented a single-agent reactive execution model that allows the agent to execute its plan while it is simultaneously calculating a repairing structure for repairing potential failures in the subsequent portion of the plan. This anticipatory behaviour grants reactivity and a quick responsiveness: if a failure occurs, there will be always available a repairing structure that comprises a restricted representation of potentially reachable world states and with which the agent will be able to promptly find a recovery plan. This model, embedded in the PELEA architecture, ensures a continuous and uninterruptedly flow of the execution agent, thus saving the agent from the need to resort to a planning agent.

Following our investigations in single reactivity, we are interested in exploring the extension of the model to a multi-agent context for collaborative repair where at least two agents participate in the final solution. The multi-agent scenario differs considerably from a single-agent context because agents cannot foresee in advance which other agent may claim for help and, therefore, the repairing structures of the agents do not contain helpful information for assisting the agent. This way, the mission of an agent who is asked to provide help consists in creating at the time of the request a combined repairing structure that supports fixing the failure of the demanding agent as well as the adaptation of its own plan to repair the failure. Ultimately, the objective is to come up with a *Multi-Agent Reactive Planning and Execution* (MARPE) model that ensures the continuous and uninterruptedly flow of the execution agents.

Since we are working in a reactive environment where several agents execute their plans, our aim is to support two main features in the collaborative repair process:

1. **Involvement of multiple agents**: an agent can request help from one or more agents.

2. **Promptly response**: To improve responsiveness, only a maximum of two agents can intervene in the repair process, the agent that fails and the agent that provides the help. We limit the provision of assistance to a single agent for the sake of obtaining a promptly response. The more agents involved in a response, the more time needed to calculate a response. Clearly, this restricts the failures that can be reactively solved. However, if solving the failure of an agent requires more than one solver agent then it is likely to be more worthwhile to resort to replanning than slowing down the execution of multiple agents. Ultimately, the MARPE model is not as a solution to cooperatively solve all the upcoming failures but a system towards a global reactive solution.

Our MARPE model is aimed to work within a reactive context and thereby we advocate for a fast solution to solving the failure. More specifically, our primary objective is that the agent that seeks assistance can obtain a promptly solution, if possible.

This chapter is organized as follows: Section 5.1 introduces a motivating example to illustrate the different concepts that will be presented throughout the Chapter. Next, we present the architecture in which the MARPE model is embedded. Section 5.3 introduces the context and the characteristics that define the recovery process of a failure in a multi-agent environment. Section 5.4 explains in detail the collaborative repair mechanism and, finally, Section 5.5 presents some conclusions.

5.1 Motivating example

In this section, we present an illustrative example that describes the behaviour of several agents when they participate in a multi-agent repair process. The example extends the Planetary Mars scenario of Section 3.2 with two more rovers.

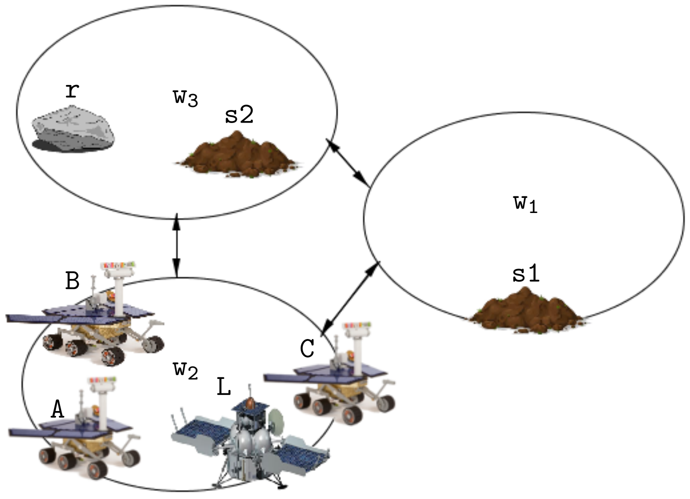

Figure 5.1: Initial state of the Mars domain motivation scenario.

Figure 5.1 shows the initial situation of a particular problem of a multi-agent version of the Mars domain scenario (see Appendix A.2) with three rovers (A, B and C). The waypoint w_2 is the initial location of lander L and the rovers, and L remains always in w_2. The mission of A is to use the microscopic camera to analyze rocks r located in w_3, communicate the results to L, and navigate to the initial position w_2. The mission of B is to analyze a soil sample s1 located in w_1, communicate the results to L from w_1, and navigate to w_2. The mission of C is to analyze a soil sample s2 located in w_3, communicate the results to L from w_3 and navigate to w_2. Rovers can only communicate if the transmission device is not disabled, i.e. the fluents ⟨trans-A,true⟩, ⟨trans-B,true⟩, and ⟨trans-C,true⟩ are present in the current world state. Likewise, they can only analyze soil or rock samples if the device to analyze is not disabled; i.e., if fluents ⟨analyze-A,true⟩, ⟨analyze-B,true⟩, and ⟨analyze-C,true⟩ hold in the current world state. On the other hand, A and B have capabilities to explore the area seeking for more soil or rock samples (fluents ⟨seek-A,true⟩ and ⟨seek-B,true⟩). Rovers have good maps to travel between two waypoints, except for rover C, which can not navigate from w_2 to w_3.

Rovers are entities that encompass a planning agent and an execution agent. The planning agent of each rover calculates a plan for its respective problem that will be executed by the associated execution agent in a shared environment (the solution plan of each rover is shown in Appendix A.2.4).

While planning agents generate the plans, rovers publicize each other how they can contribute if a failure occurs over the course of the execution. Thus, agents are apprised of the **services** or capabilities provided by the other agents, which they might need to resort to if a failure occurs during the execution.

In case of a failure, the first thing an execution agent will attempt is to repair it with its own repairing structure, as we explained in Section 3.4.2. Whenever self-repairing is not possible, the agent will seek assistance from other agents in order to attempt solving the failure collectively.

Given our Mars scenario, let us assume that A fails when executing the action (Analyze A r w_3). The failure produces two alterations in the current world state: i) rover A has no longer the capability to analyze rocks and ii) its current location is w_1 instead of w_3; i.e., the fluents ⟨analyze-A,false⟩ and ⟨loc-A,w_1⟩ hold in the world state. This failure prevents A from analyzing the rock.

Let's assume rover A is not able to fix the failure with its single repair mechanism and so it requests help from rover B. Through the publicized services, rover A knows that rover B is able to analyze the rock r and send the results of the analysis back to him; i.e., B can reach the fluent ⟨have-A,r⟩. Rover B receives A's request and tries to find a recovery path that gives support to rover A as well as keeping the course of the execution of its current plan window. Assuming that rover B finds a recovery path, when the actions of the recovery path are executed, rover A will dispose of the rock analysis. Consequently, the only remaining actions that A needs to execute are (1) navigating from w_1 to w_3 (2) communicating the results of the analysis to L and (3) navigating to the destination waypoint w_2.

5.2 Architecture

In this section, we present the architecture of our MARPE model. More specifically, we extend the architecture presented in Chapter 3 with two new mechanisms:

1. *Publicization services:* through this mechanism, an agent informs the rest of agents of the fluents it can achieve; i.e., how it can contribute and assist others in the case of a potential failure. Agents publicize their services before plan execution, when planning agents are calculating the initial solution plan for each execution agent.

2. *Collaborative repair process:* we augment the capabilities of the Reactive Planner with a repair mechanism that allows execution agents to solve plan failures collectively. For this purpose, agents use the services publicized by the other agents in order to know who is a potential contributor if a failure occurs.

The collaborative repair process, which is the objective of this chapter, relies upon a recovery process that extends the *Reactive Planning and Execution* (RPE) model presented in Chapter 3 to a multi-agent environment. Figure 5.2 shows the control flow of the architecture of our MARPE model. We incorporate a new coordination level between the execution agents through the two functionalities mentioned above: the publicization services and the collaborative repair mechanism.

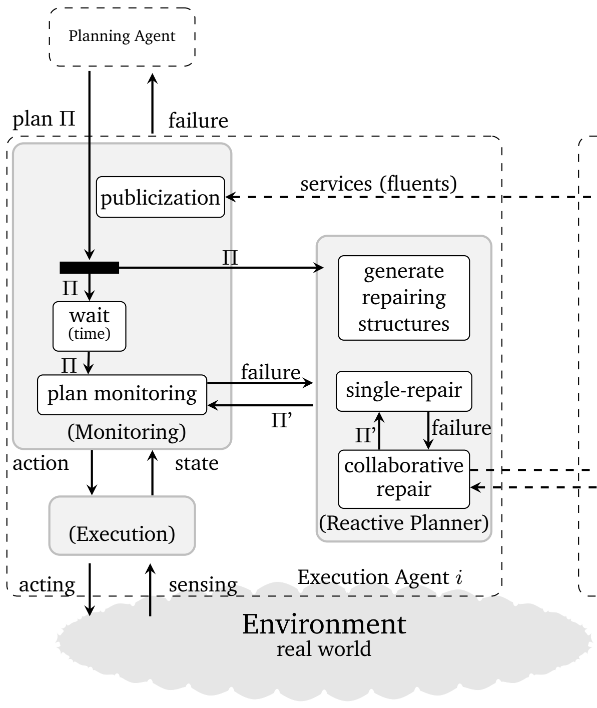

Figure 5.2: Architecture of the multi reactive planning and execution model.

Given an execution agent i registered in the system, after soliciting the initial solution plan to its planning agent (planner), agent i will publicize the **services** information through a broadcast message to the rest of execution agents (boxes *publicization* in Figure 5.2). These services are the fluents that appear in the effects of the actions of the planning task of i. Agent i publicizes its achievable fluents so that the rest of agents are aware of the capabilities of i. Then, the recipient agents remove the fluents that are needless to them and record agent i as a potential helper agent. For example, if agent i publicizes the fluent $\langle v, p \rangle$, agent j will ignore such a fluent if $\langle v, p \rangle$ is not defined in the preconditions of its actions nor in the goals of its planning task. Otherwise, j will register that the fluent $\langle v, p\rangle$ is achievable by agent i so that agent j may request agent i to repair the fluent $\langle v, p\rangle$ in case of a failure in the plan of agent j.

In our motivating example, rover A publicizes with an asynchronous broadcast message the following fluents to rover B and rover C:

- ⟨locs-s1,w1⟩, ⟨locs-s1,w2⟩, ⟨locs-s1,w3⟩, ⟨locs-s2,w1⟩, ⟨locs-s2,w2⟩, ⟨locs-s2,w3⟩, ⟨locs-r,w1⟩, ⟨locs-r,w2⟩, and ⟨locs-r,w3⟩; these fluents denote that A can help seek more soil or rock samples in any waypoint.
- ⟨loc-A,w1⟩, ⟨loc-A,w2⟩ and ⟨loc-A,w3⟩; meaning that A can navigate to w1, w2, or w3.
- ⟨have-A,s1⟩, ⟨have-A,s2⟩, ⟨have-A,r⟩, ⟨have-B,s1⟩, ⟨have-B,s2⟩, ⟨have-B,r⟩, ⟨have-C,s1⟩, ⟨have-C,s2⟩, ⟨have-C,r⟩; these fluents indicate that A can achieve the results of the analysis of soil and/or rock samples for rover B and C; this capability is also applied to itself.
- ⟨comm-r-w1,true⟩, ⟨comm-r-w2,true⟩, ⟨comm-r-w3,true⟩, ⟨comm-s-w1,true⟩, ⟨comm-s-w2,true⟩, and ⟨comm-s-w3,true⟩; these fluents represent that rover A is able to communicate the results of the analysis to the lander L from any waypoint.

Rovers B and C receive the services communicated by A and filter out the fluents that are useless to them; e.g., the fluents that represent something that A can achieve by itself (navigate to any waypoint) or the fluents that are not defined in the preconditions of their actions. Likewise, B and C publicize the same services as rover A, except that rover C is not able to achieve fluents related to finding new samples of soil or rock because rover C does not have capabilities to seek more samples.

In our approach, agents generate the service information automatically and receive updated information at any time during the *plan execution*. For instance, if rover A looses the capability of communicating the results of the analysis then rover

A will publicize a new service information, excluding any fluent of the form $\langle$comm-s_i-w_i,true$\rangle$, in which s_i is a soil or rock sample and w_i is a waypoint; and excluding also the fluents $\langle$have-B,$s_i\rangle$ and $\langle$have-C,$s_i\rangle$, which are achieved through the action Comm-rover that communicates the results of the analysis to the other rovers (B and C in this example).

Communicating service information in a multi-agent environment is common practice. In some approaches, the transmitted service information is the relationships that agents discover during the planning process such that agents learn how their plans may affect or be affected by other agents [65]. In our case, however, service information is used at execution time to inform agents who they can address in case of a failure.

In a multi-agent system where all agents work in the same environment, the solution returned by the planning and repair methods (*single-repair*, *collaborative repair*, and the *Planning Agent* in Figure 5.2) may interfere with the actions of the plan of any other agent (for instance, in our rovers scenario, an interference will occur if both rover A and rover B have a plan to analyze the rock r at the same time). In this kind of situations, we advocate for solving the interference during execution with our MARPE model whenever the monitor module detects the interference.That is, instead of detecting and solving conflicts between plans (interferences) at planning time, we relegate these to execution time, thus making an interference become a potentially failure when plans are executed at runtime, and hence solving the failure with the collaborative repair of our MARPE model [1].

Next section explains the context and characteristics where the **collaborative repair** takes place. The collaborative repair is explained in detail in Section 5.4.

[1]A different behaviour of our MARPE model is to not allow recovery plans which interfere somehow with the actions of other plans under execution. In this case, the agent activates a verification process that checks whether the recovery solution negatively affects any other plan being currently executed; if so, the agent does not accept the recovery plan as valid. The conflict verification process is explained in Appendix G

5.3 Defining the formal context

In this section, we explain the context and the characteristics that define the recovery process of a failure in a multi-agent environment. The key purpose of the collaborative repair is to let agents assist others to repair a failure, if possible, in order to ensure a continuous and uninterruptedly execution. If agents do not reach an agreement to collaborate to fix together a particular failure, then the failed execution agent calls its planning agent to obtain a new plan.

This section is organized as follows. First, we formally define the planning tasks of multiple agents that work in a common environment and the existing connection between the knowledge of the agents. Next, we explain the context of a plan failure in an unpredictable environment where several execution agents are working together.

5.3.1 Planning tasks of the agents

As we described in the motivating example of Section 5.1, execution agents are independent to each other and they are assigned a plan that they must execute to reach the goals of their tasks. All the agents operate in the same environment and they can share certain pieces of information depending on the type of agent. Specifically, the collaborative repair is built on the grounds that agents although independent handle some common knowledge and information of the environment (*publicization* services), which is the key issue to come up with a collaborative repair.

Formally, a multi-agent planning task is a collection of independent planning tasks (see the definition of planning task in Definition 3.1), one per agent. Let $\mathcal{AG}$ be a finite non-empty set of agents $\{1, \dots, n\}$; the planning task of an agent $i \in \mathcal{AG}$ is defined as $\mathcal{P}^i = \langle \mathcal{V}^i, \mathcal{A}^i, \mathcal{I}^i, \mathcal{G}^i \rangle$, where:

- $\mathcal{V}^i$ is the set of state variables known to agent i. Given two agents i and j with their respective $\mathcal{V}^i$ and $\mathcal{V}^j$, there may be variables that are shared by the two agents, i.e. $v \in \mathcal{V}^i \cap \mathcal{V}^j$. During the publicization service, recipient agents store the variables shared with other agents. In our motivating example, the set of agents $\mathcal{AG}$ is $\{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$. The agent `A` has the set $\mathcal{V}^{\mathtt{A}}$ that contains variables like `trans-A` or `loc-L`; the agent `B` has the set $\mathcal{V}^{\mathtt{B}}$ that contains variables like `trans-B` or `loc-L`; and the agent `C` has the set $\mathcal{V}^{\mathtt{C}}$ that contains variables like `trans-C` or `loc-L`. Hence, the three agents share the variable `loc-L`.

- $\mathcal{A}^i$ is the set of capabilities (planning actions) of a given agent i. Given two agents i and j, agent i can modify with its capabilities the value d of a variable v such that $v \in \mathcal{V}^i \cap \mathcal{V}^j$. For this reason, during publicization, an agent i publicizes the fluents $\langle v, d \rangle$ that it can achieve with its planning actions, i.e. $\langle v, d \rangle \in eff(a)/a \in \mathcal{A}^i$, so the rest of agents are informed of the capabilities of i. For instance, in our motivating example, the rovers `A` and `B` have the same capability to Seek for more soil or rock samples and thereby the two rovers can modify the value of the variables `locs-s1`, `locs-s2` or `locs-r`. As another example, rover `A` can change the value of the variable `have-B` by executing the action Comm-rover that offers `A` the capability to communicate the analysis of a given sample to another rover.

- $\mathcal{I}^i$ is the subset of fluents of the initial state $\mathcal{I}$ that are visible to i. Although the initial state of the world is the same for all the agents, the visibility that each agent i has of the world state is limited to its state variables $\mathcal{V}^i$. For instance, `A` does not know the variables `trans-B` and `trans-C` that represent a transmission device located in the rovers `B` and `C`, respectively. Hence, fluents `trans-B` and `trans-C` will not be part of $\mathcal{I}^{\mathtt{A}}$.

- $\mathcal{G}^i$ is the set of fluents that define the goals of the task $\mathcal{P}^i$. In our multi-agent planning task, the agents' goals are disjoint sets; i.e., $\mathcal{G}^i \cap \mathcal{G}^j = \emptyset$. An example of a fluent goal of rover `C` is $\langle \mathtt{loc\text{-}C}, \mathtt{w}_2 \rangle \in \mathcal{G}^C$.

A multi-agent scenario works as follows: each planning agent calculates a plan

Π^i for the goals $\mathcal{G}^i$ of the task $\mathcal{P}^i$ and then the associated execution agent will execute Π^i. Given that the goals of the planning tasks are disjoint sets and that planning agents independently compute a plan for each task, there do not exist dependencies among the actions of the computed plans. However, since execution agents operate in a common environment, as shown in the motivating example of Section 5.1, conflicts (failures) between the agents' plans may arise at execution time.

5.3.2 Plan failures in dynamic environments

The RPE model presented in Chapter 3 draws upon the construction of the regressed plan for a sequence of actions or plan Π (Definition 3.3). Given a planning task $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ and a plan Π that solves $\mathcal{P}$, the chronologically sequence of partial states is obtained by regressing the task goals $\mathcal{G}$ through the actions of Π. That is, each partial state of the regressed plan represents the minimal set of fluents that must be true in the current state at each point to ensure that $\mathcal{G}$ is accomplished.

In highly dynamic and unpredictable environments, where failures may frequently occur, it is preferable to focus exclusively on restoring the most immediate actions regardless whether the repair affects the achievement of the task goals $\mathcal{G}$ since obtaining a fully executable plan is usually unaffordable. Besides, guaranteeing the fulfilment of $\mathcal{G}$ can be regarded as waste effort given that the occurrence of future failures will most likely affect again $\mathcal{G}$. Thus, conducting a repair towards the achievement of $\mathcal{G}$ may not be very sensible. Let us consider, for example, an execution agent which has a 20-action plan to execute and that a failure occurs in the second action of the plan. The most priority task is to repair the second action so as to promptly resume the plan execution even at the expense of some actions of the rest of the plan becoming non-executable. This will obviously bring about a failure later in the plan, which will be handled as a failure provoked by any other factor. Our claim is that our model is specifically designed to account for failures at runtime, no matter the cause, and so it seems reasonable to delimit the repair context to a subset of the actions of Π instead of applying a time-consuming process that considers the entire Π.

This new view of a failure context may cause a later flaw in a fluent of Π since the recovery plan will not ensure now the fulfilment of the minimal set of fluents to achieve $\mathcal{G}$. However, it does foster reactivity as the RPE model will now operate with the fluents of the root node that only account for the selected plan window or *repair context*. Moreover, limiting the repair context to the current window will also increase the chance of agents helping each other since now agents only need to ensure the executability of the actions contained in their current plan windows. Likewise, helper agents that get involved in a collective repair will work to fix the failure of the assisted agent considering only the immediate goals of its current plan window instead of the task goals, $\mathcal{G}$.

In order to account for the aforementioned observations, we need to change the calculation of the root node of a plan window. Now, the root node associated to a plan window will no longer contain the necessary fluents to ensure the achievement of $\mathcal{G}$, but only the fluents that represent the actual purpose of executing the actions in the plan window. In other words, given a sub plan $[a_i, \ldots, a_l]$ of a plan Π $= [a_1, \ldots, a_n]$, we have that:

1. in a single-agent environment, the root node of the repairing structure associated to the plan window $[a_i, \ldots, a_l]$ is a regressed partial state that contains the minimal set of fluents that must hold in the world state S at time $t = l + 1$ to ensure the sequence of actions $[a_{l+1}, \ldots, a_n]$ is executable in S.

2. in a multi-agent environment, the root node of the repairing structure associated to the plan window $[a_i, \ldots, a_l]$ is a non-regressed partial state which contains the fluents that denote the ultimate purpose of executing the action sequence $[a_i, \ldots, a_l]$. We refer to this set of fluents as *the purpose of the plan window*. The purpose of a plan window is the set of fluents produced by

the actions of the sequence $[a_i, \ldots, a_l]$ that are not used by a later action of the subsequence to produce another fluent. More specifically, if an action a_j generates a fluent f which is used by a further action a_k in the sequence to produce a fluent f', the fluent f is not part of the purpose of the plan window but a supporting fluent to generate f'.

The following section explains the procedure to compute the root node of a repairing structure in a multi-agent context.

5.3.3 Root node of a repairing structure

In order to calculate the fluents of the root node of a repairing structure in a multi-agent context, we define the *forward function* similarly to the state transition function defined in Equation 3.3.1. The only exception is that now the root node will only comprise the fluents that represent the *purpose of the plan window* but not the entire world state. We will refer to the root node generated with the forward function as a *forward state*.

Let $[a_i, \ldots, a_l]$ be an action sequence and F the forward state that contains the fluents that denote the purpose of $[a_i, \ldots, a_l]$. We calculate the state F as follows:

$$F = forward([a_i, \ldots, a_l]) := \bigcup_{j=i}^{l} eff(a_j) \setminus \bigcup_{j=i+1}^{l} pre(a_j) \tag{5.3.1}$$

F contains the fluents reached through the execution of $[a_i, \ldots, a_l]$, excluding the fluents that are used as a support to produce a subsequent fluent. In terms of planning, and considering the order relationships given by the plan, we are excluding the fluents f involved in a causal link of the form $[a_i \xrightarrow{f} a_j]$, meaning that the precondition f of action a_j is supported by an effect of action a_i. That is, if $f \in eff(a_i)$ and $f \in pre(a_j)$, $j > i$, then the two actions together with the fluent f form the causal link $[a_i \xrightarrow{f} a_j]$. Note that the forward function of Definition 5.3.1 includes the fluents of $eff(a_i)$ that are not used by the actions of the plan window to support the generation of a subsequent fluent.

Let us consider the plan window $[a_1, a_2]$ composed by the actions (Navigate B $\mathtt{w}_2$ $\mathtt{w}_1$) and (Analyze B s1 $\mathtt{w}_1$), respectively. The forward state associated to the plan window $[a_1, a_2]$ is $forward([a_1, a_2]) = eff(a_1) \cup eff(a_2) \setminus pre(a_2) = \{\langle$have-B,s1$\rangle\}$ (see the preconditions and effects of the actions in the *PLANNING DOMAIN DEFINITION LANGUAGE* (PDDL) domain of Appendix A.2) [2]. The effect of a_1, $\{\langle$loc-B,$\mathtt{w}_1\rangle\}$, is not included because this fluent is part of the causal link between the actions (Navigate B $\mathtt{w}_2$ $\mathtt{w}_1$) and (Analyze B s1 $\mathtt{w}_1$); the preconditions of a_2 are the fluents $\{\langle$locs-s1,$\mathtt{w}_1\rangle$, $\langle$loc-B,$\mathtt{w}_1\rangle\}$.

Given a plan window $[a_i, \ldots, a_l]$, the forward state F is the root node of the search space $\mathcal{T}$ associated to $[a_i, \ldots, a_l]$. In particular, we can affirm that $F \subseteq S$, being S the world state reached after executing $[a_i, \ldots, a_l]$ in the corresponding initial state. $\mathcal{T}$ will contain recovery paths to reach the set of fluents F that represent the purpose of executing $[a_i, \ldots, a_l]$. Depending on the selected recovery path, a failure in subsequent actions of the plan might occur.

Back to our previous example, the result of $forward([a_1, a_2])$ is $F=\{\langle$have-B,s1$\rangle\}$; i.e., the purpose of executing $[a_1, a_2]$ is that rover B has the soil s1 analyzed regardless the location of the rover. The repairing structure $\mathcal{T}$ for $[a_1, a_2]$ will comprise various recovery paths that end up analyzing the soil s1 from different waypoints. Hence, if rover B analyzes s1 from the location $\mathtt{w}_3$, the next action of the plan a_3: (Communicate B s1 L $\mathtt{w}_1$ $\mathtt{w}_2$), will not be executable. This is the main difference with respect to the regressed states used as root nodes in the repairing structures of a single-agent environment. In a multi-agent environment, the repair structure only considers the local context of the plan window under execution rather than all the fluents needed to ensure the entire plan is executable.

Additionally, representing the root node of a repairing structure as a forward

[2]The effect $\langle$locs-s1,NONE$\rangle$ of a_2 is not considered as a primary purpose of the plan window because the constant NONE is used to indicate that the variable locs-s1 has no value. More specifically, NONE denotes that the sample s1 is not located at any waypoint and thereby the fluent is not a primary purpose.

state enables more reactiveness because the repair process now puts only the focus on the fluents of the associated plan window that represent the purpose of the plan window rather than on the fluents required to execute the entire plan. We advocate the idea that any failure produced on the fluents of subsequent actions can be quickly repaired using the subsequent repairing structures. As shown in chapter 4, the evaluation of the reactive planner threw very competitive results.

5.4 Collaborative repair process

In this section, we explain in detail the Collaborative Repair process [53], in which one agent seeks assistance from other agents to repair some failure when it is unable to solve the failure with its single-agent Reactive Planner. First, we formalize the multi-agent plan repair process. Next, we present a sketch of the flow process and then we explain in detail the elements and routines of the process, which are mostly defined as multi-agent extensions of the concepts of the single-agent repair mechanism explained in Chapter 3.

5.4.1 Multi-agent plan repair formalization

We have a set of agents $\mathcal{AG}$ each executing a plan Π^i that solves its respective planning task $\mathcal{P}^i$. A detailed description of the planning task of an agent, $\mathcal{P}^i$, is given in Section 5.3.1. A plan of an agent i is given as a sequence of actions $\Pi^i = [a^i_1, \ldots, a^i_n]$ or as a sequence of partial states $\Pi^i = [G^i_0, \ldots, G^i_n]$, as shown in Chapter 3. We recall that the plans of the agents are computed independently to each other so potentially arising interferences between plans are not considered at planning time. We advocate to solve these negative interferences at execution time when the failure is detected. Thus, the contribution of this chapter is to show that dealing with failures at execution time using an appropriate multi-agent reactive model turns out to be more efficient and less costly than solving interferences during the construction of the plans with a deliberative planner.

As we mentioned in Chapter 3, when an agent reacts to a failure in its plan, the *reactive planner* actually works with a portion of the plan, called *plan window*, rather than on the entire plan of the agent. The same applies to a multi-agent context; when two or more agents get involved in a multi-agent plan repair, only the plan windows of the involved agents will intervene in the repair process. For the sake of simplicity, we will assume that $[G_0^i, \ldots, G_n^i]$ refers to the current plan window of every agent i in the system. Consequently, a failure in the agent i will always affect the first action a_1 of the plan window $[a_1^i, \ldots, a_n^i]$ of the failing agent i.

Agents individually execute their plans under a common environment. An agent makes use of its monitoring-execution system to keep track of the plan execution and uses its embedded *reactive planner* when its plan fails (see Chapter 3). In case the failure is not solvable by itself, the multi-agent repair module is activated. Within a multi-agent execution context, the failure of a plan Π^i of an agent i can arise for two reasons:

1. An unexpected event makes the action of the agent's plan become non-executable. This is the same source of failure as in a single-agent context.
2. A negative interference between the plans of two or more agents renders the plan of an agent non-executable. This source of failure only occurs in a multi-agent context.

Let us suppose that the current state of the world is S, that agent i fails when the action a_1^i of its plan window $[G_0^i, \ldots, G_n^i]$ is not properly executed and that i is not able to repair the failure by itself. Agent i then seeks assistance from another agent to recover the state G_0^i so as to be able to execute a_1^i and the remaining actions of its plan window; otherwise it will aim to recover G_1^i in order to execute a_2^i; and so on. That is, agent i will try to successively find an agent capable of restoring one of the partial states of its plan window.

Through the publicization services, agent i is aware of the potentially helper agents to restore the failed fluents of a state G^i_t. Assuming agent j is a candidate to restore a partial state G^i_t of i's plan window, the repair process is composed of two stages:

1. Agent j creates a new planning task that takes into account the current world state S, the partial state G^i_t to be reached of agent i and the last partial state G^j_m of its current plan window $[G^j_0, \ldots, G^j_m]$. Following a similar procedure as explained in [61], agent j creates a new planning task of the form $\langle \mathcal{V}^j, \mathcal{A}^j, \mathcal{S}^j, G^{ji} \rangle$ where:

   - $\mathcal{S}^j \in S$
   - $\mathcal{A}^j$ is the set of actions of the agent j
   - $G^{ji} = G^j_m \cup G^i_t$

   Note that the agent j uses its set of actions $\mathcal{A}^j$ to restore the flawed fluents and thus help agent i resume the execution of its plan window. The new partial state G^{ji} results from the combination of the target partial state of i (G^i_t) and the last partial state of the current plan window of j (G^j_m). The output of this stage is a recovery solution plan that is sent to agent i.

2. Let us assume that the recovery solution returned by agent j that restores the flawed fluents of agent i is $[G^j_0, \ldots, G^j_t]$, or equivalently, $[a^j_1, \ldots, a^j_t]$. Agent i needs now to find a plan that complies with $[a^j_1, \ldots, a^j_t]$ and enables to resume the execution of its plan window. To this end, agent i creates new planning task that takes into account its original planning task $\mathcal{P}^i$, the partial state G^i_t to reach, the current world state S and the recovery solution received from agent j. Following the same procedure as with agent j [61], we modify the planning task $\mathcal{P}^i$ to create a new planning task of the form $\langle \mathcal{V}^i, \mathcal{B}^{ij}, \mathcal{S}^i, G^{ij} \rangle$ where:

- $\mathcal{S}^i \in S$
- $\mathcal{B}^{ij}{=}\mathcal{A}^i \cup \mathcal{B}^j$ / $\mathcal{B}^j{=}\{a : a \in [a^j_1, \ldots, a^j_t]\ \}$
- $G^{ij}{=}G^i_t \cup G^j_t$

$\mathcal{S}^i$ is the subset of fluents of the world state S that are visible to i. $\mathcal{B}^{ij}$ results from the combination of the set of actions of i ($\mathcal{A}^i$) and the actions of the plan received from j. The set of actions in $\mathcal{B}^{ij}$ will be used by agent i to find a plan that resumes the execution of its plan window while conforming with the order relationships of the recovery plan of j. G^{ij} is the partial state that results from combining the target partial state of i, G^i_t, and the last partial state, G^j_t, of the solution returned by the agent j.

5.4.2 Outline of the workflow

Whenever an execution agent is not able to solve a failure with its Reactive Planner, it will invoke the Collaborative Repair process to find out whether any agent in the environment can help it out to restore the execution of its plan. In this case, the execution agent activates the collaborative process (pictured by the box labeled as *collaborative repair* in the Execution Agent i of Figure 5.2), which initiates a communication and information exchange procedure with the other agents. In our model, agents are always willing to help whenever this is possible. That is, we assume altruistic agents.

Algorithm 3 shows the general flow of the Collaborative Repair mechanism. In the following, we explain the flow between a requester agent i and a helper agent j when the Collaborative Repair process is activated. Agent i has a repairing structure associated to its plan window under execution $[a^i_1, \ldots, a^i_n]$ or, equivalently, to the partial states $[G^i_0, \ldots, G^i_n]$, where G^i_n is the root node of the plan window. Agent j has a repairing structure associated to its plan window under execution $[a^j_1, \ldots, a^j_m]$ or, equivalently, to the partial states $[G^j_0, \ldots, G^j_m]$ where G^j_m is the root node. The root nodes are forward states as exposed in Definition 5.3.1. Let us suppose that a

```
                    requester agent i                                  helper agent j
plan window:        $[G^i_0, \dots, G^i_n]$                            $[G^j_0, \dots, G^j_m]$
 1: for $G^i_t \in [G^i_0, \dots, G^i_n]$ do
 2:     F $\leftarrow$ fluents that fail in $G^i_t$ and $i$ cannot achieve
 3:     AG $\leftarrow$ agents that achieve all the fluents in F
 4:     if AG $\neq \{\}$ then
 5:         $\Pi^{\mathsf{AG}} \leftarrow \{\}$                    ▷ solution plans
 6:         for $j \in$ AG do
 7:             send $G^i_t$ to $j$
 8:                                                    $G' \leftarrow$ fluents $\langle v, p\rangle \in \{G^i_t \cap eff(a) \forall a \in \mathcal{A}^j\}$
 9:                                                    $G^{ji} \leftarrow$ Create_Joint_Partial_State($G^j_m$, $G'$)
10:                                                    $\mathcal{T} \leftarrow$ Helper_Joint_Search_Space($G^{ji}$, $\mathcal{A}^j$, $S^j$)
11:                                                    $\Pi^j \leftarrow$ Iterative_Search_Plan($\mathcal{T}$, $S^j$, $[G^{ji}]$)
12:                                                    send $\Pi^j$ to $i$        ▷ Otherwise, send reject message
13:             $\Pi \leftarrow$ fluents $\langle v, p\rangle \in \Pi^j / v \in \mathcal{V}^i$
14:             $\Pi^{\mathsf{AG}} \leftarrow$ insert $\Pi$
15:         end for
16:         for $\Pi \in \Pi^{\mathsf{AG}}$ do
17:             $\Pi^i \leftarrow$ Requester_Joint_Plan($\Pi$, $G^i_t$, $\mathcal{A}^i$, $S^i$)
18:             if $\Pi^i \neq \{\}$ then
19:                 return form multi-reactive solution
20:         end for
21: end for
```

Algorithm 3: General flow of the Collaborative Repair process.

precondition of a^i_1 fails in the current world state S^i and agent i is not able to find a recovery path with its embedded Reactive Planner, as explained in Chapter 3. In this case, agent i activates its collaborative mechanism:

1. Agent i iterates through the partial states of its plan window $[G^i_0, \dots, G^i_n]$. At each iteration, we will refer to the partial state under study, G^i_t, as the target partial state of agent i (line 1 of Algorithm 3). Agent i identifies the flawed fluents of G^i_t which are not attainable by itself (set F in line 2). Then, through the publicized services, it retrieves the set of agents AG that are capable of achieving the fluents in F (line 3). Subsequently, it initiates a process to find one agent of AG that is able to return a recovery path to achieve F (lines 4 to 6). Agent i sends the whole partial state G^i_t to agent j (line 7) because j must repair F $\in G^i_t$ while considering all the fluents that are necessary for i to continue with the execution of the remaining actions of its plan window from G^i_t.

2. With the received information, agent j performs the following operations:

   (a) it retrieves from G_t^i the set of fluents G' that are modifiable with the effects of its planning actions (line 8 of Algorithm 3). Particularly, the set G' will contain fluents that belong to one of these groups: a) fluents that fail in G_t^i and agent i **cannot** achieve (set of fluents F); b) fluents that fail in G_t^i and agent i **can** achieve; and c) fluents in G_t^i that have not failed.

   (b) it creates a new partial state G^{ji} as the result of the combination of the set G' and its root node G_m^j (line 9),

   (c) starting from the G^{ji}, it creates a new repairing structure that encodes a solution plan that fixes the flawed fluents of i as well as considering the execution of its own plan (lines 10 and 11).

   (d) if a solution is found, the solution is sent to agent i (line 12); otherwise, agent j sends a *reject message* indicating it cannot repair the failure.

3. In case agent j returns a solution Π^j, agent i retrieves from this solution the fluents that stem from variables of $\mathcal{V}^i$ and stores the filtered solution (lines 13 and 14).

4. Once agent i collects the solutions of all the agents in AG that achieve the set of fluents F, it iterates over each solution (line 16) until it finds a returned plan that can be properly adapted to resume the execution of its own plan, if any; this is so because agent i may require to execute some extra actions in order to reach other flawed fluents of G_t^i that are achievable by itself (note that it may be possible that the solution from agent j achieves all the fluents that fail in G_t^i, in which case the only operation for agent i is to wait until the execution of the solution plan of agent j finishes). The loop stops when either i) agent i finds a valid solution (lines 17-19) or ii) there are not more solutions to iterate. Assuming that agent i accepts the solution of one agent, we say the two agents have reached an agreement to collaborate together,

thus forming a multi-reactive solution. This agreement ensures a collaborative process between the two agents.

In summary, the Collaborative Repair mechanism enables solving failures produced in short portions of the plan. The requester agent uses the service information publicized by other agents in order to know the potential helper agents that can contribute to reach a partial state of its current plan window. Reaching this partial state will allow the requester agent to fix the failure and resume the execution of its current plan. The potential helper agents search for a solution plan generating a new combined repairing structure. The solution plan of the potential helper agents gives support to the requester agent without altering the purpose of execution of its plan window. If the helper agent finds a solution recovery plan that repairs the failure, the solution is sent to the requester agent, which will decide whether or not the solutions is acceptable. If an agreement is reached, the two agents will jointly execute their respective plans. When this happens, we say the two agents form a **multi-reactive solution**.

5.4.3 General concepts

For explaining the details of the Collaborative Repair mechanism, we will use two plans, presented in Appendix A.2.4, throughout this section. The two plans are associated to rovers A and B from our motivating example. Let us assume that the first two actions of each plan have already been executed and that their plan windows are updated accordingly to $[a_3^{\mathtt{A}}, \ldots, a_l^{\mathtt{A}}]$ and $[a_3^{\mathtt{B}}, \ldots, a_m^{\mathtt{B}}]$, which define the sequences of partial states $[G_2^{\mathtt{A}}, \ldots, G_l^{\mathtt{A}}]$ and $[G_2^{\mathtt{B}}, \ldots, G_m^{\mathtt{B}}]$, respectively. Subsequently, agent A fails (requester agent) when executing the action $a_3^{\mathtt{A}}$=(Communicate A r L $\mathtt{w}_3$ $\mathtt{w}_2$). This action fails because rover A lacks the results of the analysis of the rock and its current location is $\mathtt{w}_1$ instead of $\mathtt{w}_3$. That is, the fluents $\langle$have-A,r$\rangle$ and $\langle$loc-A,$\mathtt{w}_3\rangle$ are not true in the current world state. In addition, we will also assume that agent A loses the capability of analyzing rocks; i.e. the fluent $\langle$analyze-A,false$\rangle$ holds in the world state. Then, agent A activates its collaborative mechanism:

1. The first iteration of the external loop of Algorithm 3 starts with plan window $[G_2^{\mathtt{A}}, \ldots, G_l^{\mathtt{A}}]$ and $G_t^{\mathtt{A}}$ set to $G_2^{\mathtt{A}}$, which contains the fluents $\{\langle$loc-L,w$_2\rangle$,$\langle$loc-A,w$_3\rangle$,$\langle$link-A-w$_3$-w$_2$,true$\rangle$,$\langle$trans-A,true$\rangle$,$\langle$have-A,r$\rangle\}$. These are the fluents needed to execute the actions (Communicate A r L w$_3$ w$_2$) and (Navigate A w$_3$ w$_2$).

2. Agent A identifies the set of fluents that fail in $G_2^{\mathtt{A}}$ and are not achievable by itself; F=$\{\langle$have-A,r$\rangle\}$, meaning that agent A lacks the results of the analysis of the rock r; i.e., the fluent $\langle$have-A,NONE$\rangle$ holds in the current world state. Although the fluent $\langle$loc-A,w$_3\rangle \in G_2^{\mathtt{A}}$ has also failed, it is not included in F because rover A can repair this fluent. Note also that the failing fluent $\langle$analyze-A,true$\rangle$ is not in F either because this fluent is not in $G_2^{\mathtt{A}}$; that is, this fluent is not needed to execute the actions (Communicate A r L w$_3$ w$_2$) and (Navigate A w$_3$ w$_2$).

3. agent A retrieves the set of agents AG capable of achieving the set of fluents F: AG=$\{$B$\}$.

Subsequently, the whole partial state $G_2^{\mathtt{A}}$ is sent to rover B.

5.4.3.1 Joint partial state

A joint partial state combines the fluents of the partial states of two different agents, i and j. The resulting state must not contain duplicated fluents. We define Create_Joint_Partial_State(G_i, G_j) as the function that given any two partial states, G_i and G_j, generates a new one as the union of the fluents of G_i and G_j with non-repeated elements (Definition 5.4.1). We apply this process iff no conflict exists between the fluents of G_i and G_j (Definition 3.4.1):

$$\texttt{Create\_Joint\_Partial\_State}(G_i, G_j) := \begin{cases} G_i \cup G_j & : \text{ if } \neg conflict(G_i, G_j) \\ undefined & : \text{ otherwise} \end{cases} \tag{5.4.1}$$

In our example:

1. rover B receives $G_2^{\texttt{A}}$ and selects the fluents modifiable with its planning actions, discarding the fluens from $G_2^{\texttt{A}}$ that it cannot affect in any manner. Thus, we have G'={⟨have-A,r⟩}.

2. the root node of B is $G_m^{\texttt{B}}$={⟨loc-B,w$_2$⟩, ⟨comm-s1-w$_1$,true⟩}; thus, rover B calls Create_Joint_Partial_State($G_m^{\texttt{B}}$, G') and obtains the new partial state $G^{\texttt{BA}}$={⟨have-A,r⟩,⟨loc-B,w$_2$⟩,⟨comm-s1-w$_1$,true⟩}. This new partial state is simply the union of the fluents of G' and $G_m^{\texttt{B}}$ because no conflict exists between {⟨have-A,r⟩} and the fluents in $G_m^{\texttt{B}}$.

The partial state $G^{\texttt{BA}}$ will be the root node of a joint repairing structure that will comprise the actions needed for rover B to help rover A recover from its failure.

5.4.3.2 Joint Search Space of a helper agent

Algorithm 4 shows the procedure Helper_Joint_Search_Space for a helper agent j to create a new combined repairing structure using a joint partial state G^{ji} as the root node. The goal of this function is to create a search space that comprises the choices to repair the failed plan of the requester agent alongside the plan execution of the helper agent.

The algorithm receives three input parameters. The first parameter is the joint partial state G^{ji}; the second one is the set of planning actions of the agent, $\mathcal{A}^j$; and the third parameter is its current world state S^j. Agent j updates the current state S^j by applying Definition 3.3.1, $S^j = result(S^j, a)$, with the first action a of its plan window, the action that agent j is currently executing. For instance, in our example, when rover B receives the request from rover A, its plan window is $[a_3^{\mathtt{B}}, \ldots, a_m^{\mathtt{B}}]$, so B is executing the action $a_3^{\mathtt{B}}$ = (Communicate B s1 L $\mathtt{w}_1$ $\mathtt{w}_2$). The collaborative repair of rover B operates in parallel with its plan execution, which means that rover B needs to take into account the execution of the action $a_3^{\mathtt{B}}$ during the repair process. Therefore, rover B updates its current state $S^{\mathtt{B}}$ with the simulation of the execution of the action (Communicate B s1 L $\mathtt{w}_1$ $\mathtt{w}_2$). That is, rover B initiates the search of a repairing plan for the requester rover A assuming its current action has been already executed[3].

```
Input: Helper_Joint_Search_Space($G^{ji}$, $\mathcal{A}^j$, $S^j$)
 1: $\mathcal{Q} \leftarrow \{G^{ji}\}$
 2: $\mathcal{T} \leftarrow \{G^{ji}\}$
 3: $t_s \leftarrow$ set up to one execution cycle
 4: while $\mathcal{Q} \neq \emptyset$ do
 5:    $G \leftarrow$ extract first node from $\mathcal{Q}$
 6:    for all $\{a \in \mathcal{A}^j \mid relevant(G, a)$ is true$\}$ do
 7:       $G' \leftarrow regress(G, a)$
 8:       if $G' \notin \mathcal{T}$ then
 9:          if $\exists\ G'' \in \mathcal{T} \mid G'' \subset G'$ then
10:             mark $G'$ as superset of $G''$
11:          else
12:             $\mathcal{Q} \leftarrow \mathcal{Q} \cup G'$
13:          set an arc $a$ from $G'$ to $G$
14:          $\mathcal{T} \leftarrow \mathcal{T} \cup G'$
15:          if $time\_generation > t_s$ or $G' \subset S^j$ then
16:             return $\mathcal{T}$
17:    update $time\_generation$
18: return $\mathcal{T}$
```

Algorithm 4: Generate a helper joint repairing structure $\mathcal{T}$.

The procedure Helper_Joint_Search_Space works similarly to Algorithm 1. It expands the root node G^{ji} by applying the regressed transition function $regress(G^{ji}, a)$ following Equation 3.3.2 (lines 5 and 7 of Algorithm 4). This follows the classical backward construction of a planning space, where a node G is expanded (line 7) generating all possible sequences of actions that eventually reach the root node G^{ji}.

[3]In case the execution of action $a_3^{\mathtt{B}}$ fails, rover B informs rover A and resorts to replanning, and A discards the solution plan offered by rover B.

In Algorithm 1 of Chapter 3, the time limit to generate the search space $\mathcal{T}$ is implicit in the input parameter *depth of* $\mathcal{T}$ (m), which is calculated through an estimating function that ensures that a tree of depth m will be generated within a given time limit (see Section 3.4.3.1). This is so because we want an anticipatory behavior that guarantees that there will always be available a repairing structure for future actions when a failure occurs. In contrast, in a multi-agent setting, agents do not foresee which agents will request for help and, therefore, the helper agent j needs to create at runtime a combined repairing structure within a fixed time limit. More specifically, it must find a recovery solution that fixes the failure of agent i (lines 9-11 of Algorithm 3, procedures `Create_Joint_Partial_State`, `Helper_Joint_Search_Space` and `Iterative_Search_Plan`) within a maximum time of one execution cycle (line 3 of Algorithm 4).

As it was shown in the results of Chapter 4, the procedure to generate a repairing structure (line 10 of Algorithm 3) is most costly than the procedure to extract the recovery plan from $\mathcal{T}$ (line 11 of Algorithm 3), which is negligible (in average it takes $\approx$ 1.65 ms that represents 0,00165% of one execution cycle).

On the other hand, the purpose of the algorithm that creates a repairing structure for a single agent is to use this structure to repair a failure of the own agent. However, in Algorithm 4, the generation of the joint search space stops when a node whose fluents are all included in the current world state S^j is found. The reason is that in this case our aim is to create a search structure that comprises a plan for the requester agent and, therefore, the procedure stops when the structure includes a branch that encodes a repair plan for the agent.

In summary, the third parameter, S^j, and the variable t_s of Algorithm 4 are the two stop criteria for the generation of a joint search space. More specifically, the construction of $\mathcal{T}$ stops when (line 15 of Algorithm 4):

i. **the generation of $\mathcal{T}$ exceeds the time limit t_s.** This parameter ensures a time-bounded construction of $\mathcal{T}$.

ii. **there exists a node** $G' \in \mathcal{T} \mid G' \subset S^j$, in which case it means that the search space comprises a joint recovery plan from G' to G^{ji}.

In our example, rover B calls Helper_Joint_Search_Space($G^{\mathtt{BA}}$, $\mathcal{A}^{\mathtt{B}}$, $S^{\mathtt{B}}$) to create the joint repairing structure, which is graphically displayed in Figure 5.3. The figure shows the joint repairing structure with $G^{\mathtt{BA}}$ as the root node. In this case, the process stops when a node $G' \subset S^{\mathtt{B}}$ is found.

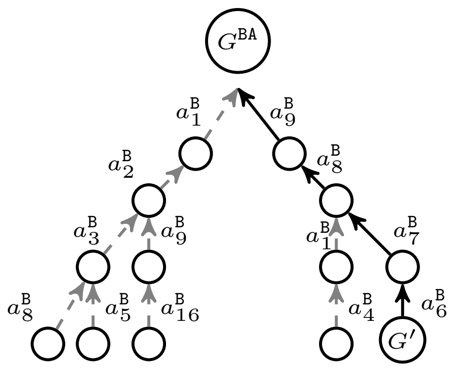

Figure 5.3: Joint repairing structure for rover B.

5.4.3.3 Recovery plan of a helper agent

Once agent j generates the joint search space $\mathcal{T}$, the next step is to extract the recovery plan that helps the requester agent i from $\mathcal{T}$ (line 11 of Algorithm 3). This is done with the same Algorithm 2 of Chapter 3.

In our example, we have that rover B calls Iterative_Search_Plan($\mathcal{T}$,$S^{\mathtt{B}}$,[$G^{\mathtt{BA}}$]) and finds the solution recovery plan shown by the path in bold of Figure 5.3, $\Pi^{\mathtt{B}} = \{a_6^{\mathtt{B}}$: (Navigate B $\mathtt{w}_1$ $\mathtt{w}_3$), $a_7^{\mathtt{B}}$: (Analyze B r $\mathtt{w}_3$), $a_8^{\mathtt{B}}$: (Comm-rover B A r $\mathtt{w}_3$), $a_9^{\mathtt{B}}$: (Navigate B $\mathtt{w}_3$ $\mathtt{w}_2$)$\}$. The solution $\Pi^{\mathtt{B}}$ navigates rover B to the location $\mathtt{w}_3$ where rover B analyzes the rock r and communicates the results of the analysis to rover A, and next, rover B navigates to its final position $\mathtt{w}_2$.

When the function Helper_Joint_Search_Space stops by the deadline t_s and a node $G' \subset S^{\mathtt{B}}$ has not been found (see stop criteria in Section 5.4.3.2), it means rover `B` is not able to find a solution recovery plan for rover `A`, in which case, rover `B` sends a *reject message* to `A` to indicate the impossibility of repairing the failure.

5.4.3.4 Construction of a joint plan by requester agent

This section explains the lines 16 to 20 of Algorithm 3. At this point, the requester agent i has one or more recovery plans returned by one or several helper agents. Agent i goes through the solutions collected in Π^{AG} (line 16) and stops when it finds one plan in Π^{AG} that can be adequately adapted to resume the execution of its plan from the target partial state G_t^i; otherwise, the list Π^{AG} is exhausted. Thus, when agent i accepts the solution of one agent j, i sends an *accept message* to agent j and they will start to collaboratively execute their solution plans in the next execution cycle (line 19). If the requester agent i does not accept any solution, it sends a *reject message* to all the helper agents and activates its Deliberative Planner as the last resource.

As it was mentioned in Section 5.4.3, the set of failing fluents in G_t^i are classified into F, the fluents to be achieved by the helper agent, and fluents that the requester agent can achieve by itself (we will refer to this set as F'). Generally speaking, adapting to the plan returned by the helper agent (line 17 of Algorithm 3) means that agent i may require to complement the plan with some additional actions to reach the fluents in F'.

The process to generate an adapted joint plan by the requester agent presents the following two scenarios:

1. **The helper agent achieves the whole set of failing fluents.** This occurs when the solution plan of the helper agent reaches all the fluents that fail in G_t^i; i.e. the fluents in F and F'.

2. **The helper agent does not achieve all the failed fluents in G_t^i.** This occurs when the solution plan of agent j achieves the fluents in F and agent i requires an additional solution plan to fix the fluents in F'.

The helper agent achieves the whole set of failing fluents. It goes without saying that the solution plan of a helper agent j achieves the fluents in F. However, it may be also the case that agent j reaches all the flawed fluents in G_t^i; i.e. it achieves both the fluents in F and F'. This situation occurs, as we exposed in item 1 of Section 5.4.2, because agent j receives the whole partial state G_t^i and tries to repair all the failing fluents in G_t^i. In other words, if agent j is able to achieve F', it will try to find a plan to repair F' as well because the agent realizes that these fluents are also necessary to resume the plan execution of agent i from G_t^i. In such a case, the adaptation plan of the requester agent i is not other than waiting until the solution plan received from agent j is completed.

Hence, assuming the solution plan offered by the helper agent j is $\Pi^j = \{a_1^j, a_2^j, a_3^j, a_4^j\}$, and that this plan achieves all the failed fluents in G_t^i, the adaptation plan of agent i will be $\Pi^i = \{a_{wait}^i, a_{wait}^i, a_{wait}^i, a_{wait}^i\}$, where a_{wait}^i is a *dummy* action that lasts one execution cycle. That is, the adaptation plan that i will execute consists solely in waiting for four execution cycles until agent j finishes the execution of its solution plan Π^j.

The helper agent does not achieve all the failed fluents in G_t^i. This second scenario occurs when the solution plan of agent j achieves only the fluents of F, in which case agent i needs to compute an adaptation plan that fixes the remaining fluents of F'.

In our example, rover A receives from rover B the solution plan $\Pi^{\mathsf{B}} = \{a_6^{\mathsf{B}}$: (Navigate B $\mathtt{w}_1$ $\mathtt{w}_3$), a_7^{B} : (Analyze B r $\mathtt{w}_3$), a_8^{B} : (Comm-rover B A r $\mathtt{w}_3$), a_9^{B} : (Navigate B $\mathtt{w}_3$ $\mathtt{w}_2$)$\}$ or, equivalently, $[G_5^{\mathsf{B}}, \ldots, G_9^{\mathsf{B}}]$, which does not accomplish all the failing fluents $\{\langle$loc-A,$\mathtt{w}_3\rangle$,$\langle$have-A,r$\rangle\}$ of G_2^{A} ($G_2^{\mathsf{A}} = \{\langle$loc-L,$\mathtt{w}_2\rangle$,$\langle$loc-A,$\mathtt{w}_3\rangle$,$\langle$link-A-$\mathtt{w}_3$-$\mathtt{w}_2$,true$\rangle$,$\langle$trans-A,true$\rangle$,$\langle$have-A,r$\rangle\}$). More specifically, Π^{B} only reaches the fluents of F (F = $\{\langle$have-A,r$\rangle\}$). Consequently, rover A needs a plan to achieve the failing fluents of F' (F' = $\{\langle$loc-A,$\mathtt{w}_3\rangle\}$); i.e., it needs a plan to navigate from location $\mathtt{w}_1$ to location $\mathtt{w}_3$. To this end, rover A performs the following operations:

1. Call `Create_Joint_Partial_State(`$G_2^{\mathtt{A}}$, $G_9^{\mathtt{B}}$`)` to create a new partial state $G^{\mathtt{AB}}$ as the result of the combination of $G_2^{\mathtt{A}}$ and $G_9^{\mathtt{B}}$, where $G_9^{\mathtt{B}}$ is the last partial state of the sequence $[G_5^{\mathtt{B}}, \ldots, G_9^{\mathtt{B}}]$ extracted from $\Pi^{\mathtt{B}}$.

2. Call `Requester_Joint_Search_Space(`$G^{\mathtt{AB}}$, $\mathcal{A}^{\mathtt{A}}$, $S^{\mathtt{A}}$, $[a_6^{\mathtt{B}}, \ldots, a_9^{\mathtt{B}}]$`)`. Starting from $G^{\mathtt{AB}}$, it creates a merged repairing structure $\mathcal{T}$ that encodes an adaptation plan to fix the remaining fluents that fail in $G_2^{\mathtt{A}}$ as well as merging its adaptation plan with the recovery plan of rover B (actions $[a_6^{\mathtt{B}}, \ldots, a_9^{\mathtt{B}}]$). The merged search space is graphically represented in Figure 5.4. The figure shows the depth of the merged repairing structure (left side of the figure); the merged repairing structure with $G^{\mathtt{AB}}$ as the root node (middle side of the figure); and the solution received from B (right side of the figure).

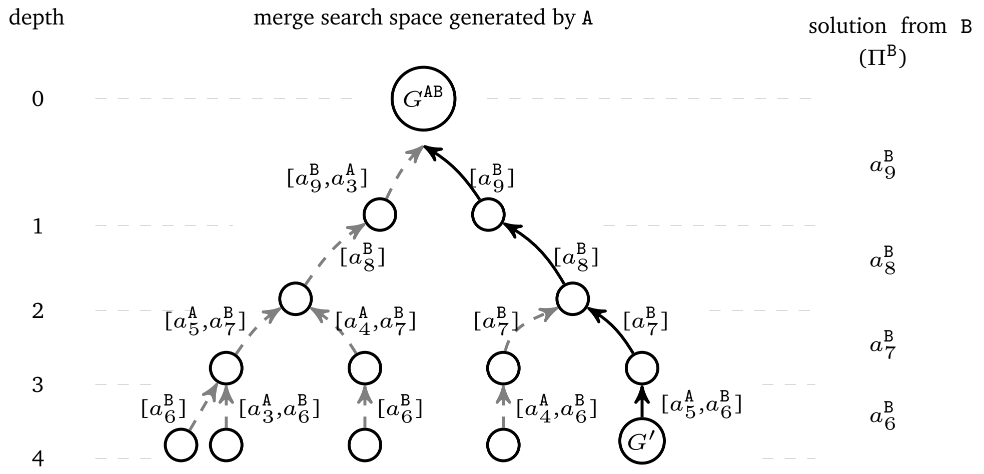

Figure 5.4: Merged Search Space

The merged structure is also generated by following the classical backward construction of a planning space, where possible sequences of actions eventually reach the root node $G^{\mathtt{AB}}$. The main characteristics of this search space are:

(a) The maximum depth of the search space is equal to the total number of actions of the solution plan of the helper agent, thus limiting the time generation of the search space. In our example, see solution from B in Figure 5.4, the solution plan of rover B has four actions, which is the maximum depth m of the merged search space of rover A. It is $m = 4$.

(b) The solution of the helper agent must be represented in all branches of the tree, one action of the plan at each tree level, always keeping the order relationships between actions. For example, in Figure 5.4, we reach the root node $G^{\mathtt{AB}}$ from two branches. In the first level of the tree (depth = 1), we represent the last action, $a_9^{\mathtt{B}}$; the subsequent down level (depth = 2), reflects the action $a_8^{\mathtt{B}}$ in both branches; the action $a_7^{\mathtt{B}}$ is represented as an outgoing (combined) arc from every node at depth=3; finally, in the last level, the action $a_6^{\mathtt{B}}$ appears as a label of the arcs coming out from every node at depth=4. This is so because rover A needs to find a solution considering the ordering of the actions in the solution plan of rover B.

(c) The construction of the merged search space $\mathcal{T}$ stops when i) the generation exceeds the maximum depth or ii) there exists a node $G' \in \mathcal{T} \mid G' \subset S^{\mathtt{A}}$, in which case it means that the merged search space $\mathcal{T}$ comprises a solution plan from G' to $G^{\mathtt{AB}}$. Obviously, this is a conflict-free solution with the recovery plan offered by rover B, thus allowing the parallel execution of the two solution plans.

3. Call `Requester_Iterative_Search_Plan(`$\mathcal{T}$, $S^{\mathtt{A}}$, [$G^{\mathtt{AB}}$]). Once rover A generates $\mathcal{T}$, the merged search space $\mathcal{T}$ is used to find an adaptation plan $\Pi^{\mathtt{A}}$ that fixes the rest of fluents that fail in $G_2^{\mathtt{A}}$, as well as reaching the last partial state $G_9^{\mathtt{B}}$ extracted from the recovery solution $\Pi^{\mathtt{B}}$ received from B.

   The procedure `Requester_Iterative_Search_Plan` works similarly to Algorithm 2 (see Section 3.4.2). In this case, the plan window of the third argument of the algorithm contains only one partial state, the root node of $\mathcal{T}$ or $G^{\mathtt{AB}}$. The procedure tries to find a node $G' \subset S^{\mathtt{A}}$ from $G^{\mathtt{AB}}$ and stops when some G' is found, in which case the path from G' to $G^{\mathtt{AB}}$ is returned. If no G'

is found, this means that rover A is not able to repair the rest of the failing fluents of $G_2^{\mathtt{A}}$.

In Algorithm 2 of Chapter 3, the returned plan is generated considering a search space that contains actions of one single agent. In contrast, the procedure `Requester_Iterative_Search_Plan` uses the merged search space which contains actions of two agents. That is, the branches of the merged repairing structure contains either one action (the one of the helper agent) or two actions (one of the helper agent and another action of the requester agent). For instance, in the branch shown in black lines of Figure 5.4.

(a) The path from the node at depth = 1 to the root node contains only an action of B, $[a_9^{\mathtt{B}}]$, meaning that rover A has to wait for one execution cycle.

(b) The path from the node at depth 4 to the upper level contains two actions, $[a_5^{\mathtt{A}}, a_6^{\mathtt{B}}]$, meaning that rover A executes its action $a_5^{\mathtt{A}}$ while rover B executes the action $a_6^{\mathtt{B}}$.

Hence, rover A calls `Requester_Iterative_Search_Plan`($\mathcal{T}$,$S^{\mathtt{A}}$,[$G^{\mathtt{AB}}$]) and returns the solution shown in black lines of Figure 5.4 as follows, $\Pi^{\mathtt{A}}$=$\{a_5^{\mathtt{A}}$: (Navigate A $\mathtt{w}_1$ $\mathtt{w}_3$), $a_{wait}^{\mathtt{A}}$: (Wait A), $a_{wait}^{\mathtt{A}}$: (Wait A), $a_{wait}^{\mathtt{A}}$: (Wait A)$\}$. The solution $\Pi^{\mathtt{A}}$ will navigate rover A to the location $\mathtt{w}_3$ and will wait three execution cycles until the plan execution of rover B finishes.

The previous iterative collaborative repair process describes the behaviour of our multi-repair mechanism. At the end of the collaborative repair process, the two agents will be cooperating together forming a reactive solution.

5.4.4 Execution of the reactive solution plan

Figure 5.5 summarizes the execution of our illustrative example with the MARPE model. The MARPE model ensures the continuous and uninterruptedly flow of the

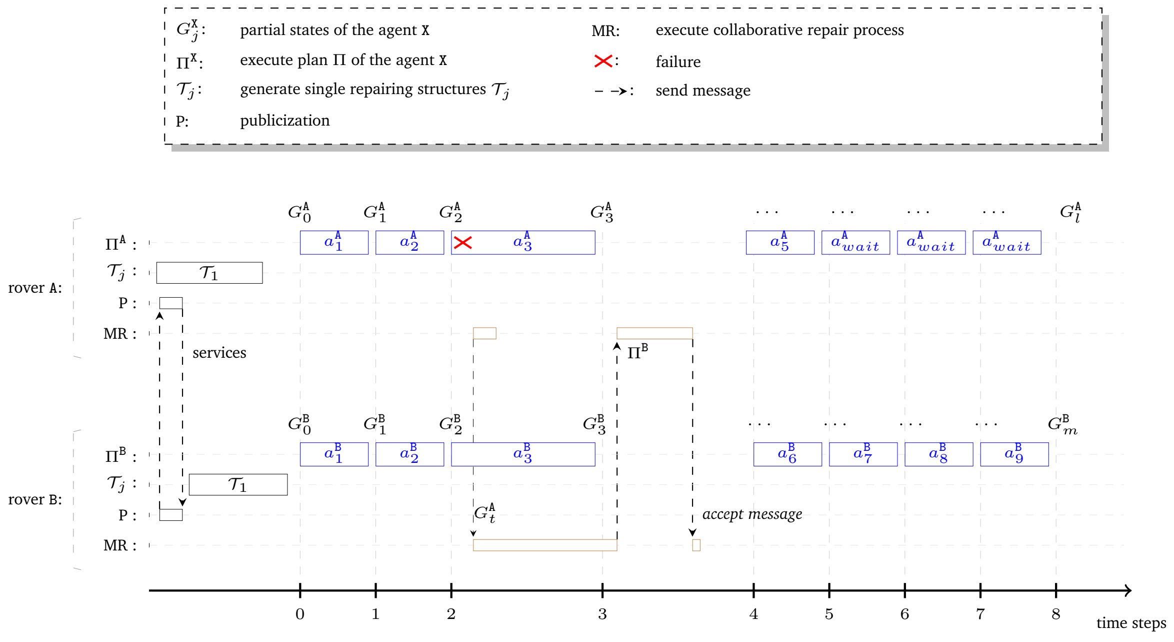
$G_j^{\mathtt{X}}$: partial states of the agent X
$\Pi^{\mathtt{X}}$: execute plan Π of the agent X
$\mathcal{T}_j$: generate single repairing structures $\mathcal{T}_j$
P: publicization
MR: execute collaborative repair process
failure
send message
rover A:
rover B:
$\Pi^{\mathtt{A}}$:
$\Pi^{\mathtt{B}}$:
$\mathcal{T}_j$:
P :
MR :
$\mathcal{T}_1$
services
$\Pi^{\mathtt{B}}$
$G_t^{\mathtt{A}}$
accept message
$G_0^{\mathtt{A}}$ $G_1^{\mathtt{A}}$ $G_2^{\mathtt{A}}$ $G_3^{\mathtt{A}}$ $G_l^{\mathtt{A}}$
$G_0^{\mathtt{B}}$ $G_1^{\mathtt{B}}$ $G_2^{\mathtt{B}}$ $G_3^{\mathtt{B}}$ $G_m^{\mathtt{B}}$
$a_1^{\mathtt{A}}$ $a_2^{\mathtt{A}}$ $a_3^{\mathtt{A}}$ $a_5^{\mathtt{A}}$ $a_{wait}^{\mathtt{A}}$ $a_{wait}^{\mathtt{A}}$ $a_{wait}^{\mathtt{A}}$
$a_1^{\mathtt{B}}$ $a_2^{\mathtt{B}}$ $a_3^{\mathtt{B}}$ $a_6^{\mathtt{B}}$ $a_7^{\mathtt{B}}$ $a_8^{\mathtt{B}}$ $a_9^{\mathtt{B}}$
0 1 2 3 4 5 6 7 8
time steps

Figure 5.5: Executing the multi-agent planning and execution model

execution agents comprising reactive planning, execution and plan monitoring. In Figure 5.5, for each rover of our example (A and B), we show:

- $\Pi^{\mathtt{X}}$: it represents the plan of the agent X.
- $G_i^{\mathtt{X}}$: it represents the partial state of the agent X that must hold at time step i.
- $\mathcal{T}_j$: it represents the repairing structure for the single repair process.
- P: it represents the publicization services
- MR: it represents the collaborative repair process execution.

In Figure 5.5, rover A fails at time step 2 when it tries to execute action $a_3^{\mathtt{A}}$. Since it is not able to repair the failure with its single reactive planner, it activates the collaborative mechanism (first small box in MR of rover A in Figure 5.5):

- Rover B receives the request from rover A when it is executing action $a_3^{\mathtt{B}}$. Simultaneously with the execution of $a_3^{\mathtt{B}}$, rover B has to find a recovery solution plan within a time limit of one execution cycle (first box in MR of rover B in Figure 5.5) with the procedures Helper_Joint_Search_Space and Iterative_Search_Plan explained in Sections 5.4.3.2 and 5.4.3.3, respectively.

  Rover B cannot execute any other action until the collaborative repair process finishes, except for the ongoing action $a_3^{\mathtt{B}}$.

- Rover A receives the recovery solution plan from rover B (second box in MR of rover A) and starts the procedures explained in Section 5.4.3.4 to accept the solution as valid. If rover A accepts the solution of rover B, it sends an *accept message* to rover B and, in the next execution cycle, the two rovers will execute their recovery solution plans (from time step 4 in Figure 5.5). Otherwise, B resumes the execution of the next action of its current plan and rover A goes to replanning.

The MARPE model guarantees that the two agents involved in the Collaborative Repair process can safely continue with the execution of their plans. However, during the execution of the collaborative solution, from time step 4 until 8 in Figure 5.5, new failures can arise, and so the agents need to deal with them within the context of the Collaborative Repair (attending the dependencies of their plans).

For example, in case the helper agent B fails and tries to repair it with the single-repair method (explained in Section 3.4), the dependencies with rover A will not be taken into account. This is because the self-repair method will attempt to achieve the fluents of B, including the failed ones, but not the fluents that B achieves for the assisted agent A within the collaborative solution. In other words, the single-repair method of B is only responsible for achieving the fluents of B but not the fluents that B has committed to agent A. Since a collaborative solution involves two tightly interdependent agents, any individual action from an agent would have a very negative impact on the other agent.

Specifically, throughout the execution of the multi-reactive solution plan, two possible failure situations can stand in regards with the two agents involved in the solution:

1. **The helper agent fails.** In this situation, the agent B fails during the execution of the reactive solution.
2. **The requester agent fails.** In this situation, the assisted agent A fails during the execution of the reactive solution.

Some important aspects should be considered in the two possible failing situations. Whenever any of the two situations arise, the agents will not apply the MARPE model as it is, because they should consider the relationships of the multi-reactive solution plan being executed. The two agents need to solve the failure without requesting help from other agents because they are already working collaboratively. In other words, if any action of the rovers A or B cannot be executed due to a plan failure, they are not allowed to ask for help to any other agent C in the environment. In contrast, if during the execution of the multi-reactive solution plan, a plan failure is provoked in the plan of a third agent C, the rover C will activate the MARPE model but with the restriction that the two agents, A and B, will reject the request of C because they are already cooperating together.

In the following, we describe briefly the process that our model performs to recover from the two failure situations that affect the helper agent and the requester agent.

In any of the two situations, for simplicity, we will say the agent activates the Collaborative Repair process with the particularity that i) it does not call at first place the single-reactive method explained in Section 3.4 and ii) it does not call the collaborative repair method from the beginning. More specifically, the failed agent performs the following processes that obviously are parts of our MARPE model.

Let us also continue with our motivating example, where the agents A and B are working together with the following plan window of their plans: [$a_5^{\mathtt{A}}$, ..., $a_l^{\mathtt{A}}$] and [$a_6^{\mathtt{B}}$, ..., $a_m^{\mathtt{B}}$] (same situation from time step 4.0 of Figure 5.5). The plans of the two agents define the sequences of partial states [$G_4^{\mathtt{A}}, \ldots, G_l^{\mathtt{A}}$] and [$G_5^{\mathtt{B}}, \ldots, G_m^{\mathtt{B}}$].

5.4.4.1 The helper agent fails

The situation occurs when during the execution of the collaborative solution the helper agent, which is reaching some fluents for the requester agent, cannot execute an action of its plan due to a failure. In such a case, the helper agent activates the Collaborative Repair to generate a multi-reactive solution that solves the failure.

The solution offered by the Collaborative Repair process should consider the fluents that the helper agent requires for the requester agent. If the failure cannot be fixed then the helper agent will not be able to reach neither its fluents nor the fluents of the requester agent. Consequently, both agent will invoke replanning.

In our example, assuming the helper agent B fails to execute the first action $a_6^{\mathtt{B}}$ (Navigate B $\mathtt{w}_1$ $\mathtt{w}_3$), the failure produces one alteration in the location of B (the current location of B is $\mathtt{w}_2$). Since B is collaborating with A, it cannot request assistance to any other external agent. Thereby, B needs to find a recovery plan that solves its failure while cooperating with A. Thus, B tries to solve the failure taking into account the current world state $\mathcal{S}^{\mathtt{B}} \in S$, the recovery solution of agent A, [$a_{wait}^{\mathtt{A}}, \ldots, a_l^{\mathtt{A}}$], or equivalently, [$G_5^{\mathtt{A}}, \ldots, G_l^{\mathtt{A}}$] [4], its root partial state $G_m^{\mathtt{B}}$ to be reached, and its original planning task $\mathcal{P}^{\mathtt{B}}$. Thus, rover B performs the following operations:

1. Call `Create_Joint_Partial_State`($G_m^{\mathtt{B}}$, $G_l^{\mathtt{A}}$). It creates a new partial state $G^{\mathtt{BA}}$ as the result of the combination of $G_m^{\mathtt{B}}$ and $G_l^{\mathtt{A}}$, where $G_l^{\mathtt{A}}$ is the last partial state of the recovery solution of agent A.

2. Call `Reschedule_Joint_Search_Space`($G^{\mathtt{BA}}$, $\mathcal{A}^{\mathtt{B}}$, $S^{\mathtt{B}}$, [$a_{wait}^{\mathtt{A}}, \ldots, a_l^{\mathtt{A}}$]). This method works similarly to the method `Requester_Joint_Search_Space` explained in

[4] Note that the action $a_5^{\mathtt{A}}$ of A does not fail and is currently in execution.

Section 5.4.3.4. Starting from $G^{\mathtt{BA}}$, it creates a merge repairing structure $\mathcal{T}$ that encodes a solution plan to recover from its failure, as well as, merging its solution plan with the current plan of rover A, actions $[a^{\mathtt{A}}_{wait}, \ldots, a^{\mathtt{A}}_{l}]$. The merge structure $\mathcal{T}$ is a search space with the same features that we explained for the Figure 5.4. The main difference of this search space is that actions of the plan of the helper agent A might be rescheduled in the new recovery plan.

If the rover B finds a new recovery plan, B informs to A, which must change and reschedule its current solution plan to the one obtained with the Requester_Joint_Search_Space procedure. Otherwise, B does not find a new recovery plan and informs with an error message to A. The error message indicates the agent A that B cannot achieve the fluents that were supported by B and, thereby, both agents will invoke their planning agent for replanning.

5.4.4.2 The requester agent fails

This situation occurs if during the execution of the collaborative solution the requester agent, which receives help from the helper agent, fails. In such a case, the requester agent activates the Collaborative Repair to generate a collaborative solution that solves the failure. The solution generated by the Collaborative Repair process should consider the solution offered previously by the helper agent. More specifically, the Collaborative Repair of the requester agent applies the same procedure explained in Section 5.4.3.4.

Let us assume the rover A fails to execute the first action $a^{\mathtt{A}}_{5}$ (Navigate A $\mathtt{w}_1$ $\mathtt{w}_3$). The failure produces one alteration in the location of A (the current location of A is $\mathtt{w}_2$). As we mentioned before, rover A cannot request assistance from other external agents because it is collaborating with B. The rover A solves the failure with the recovery solution $\Pi^{\mathtt{A}}=\{a^{\mathtt{A}}_{6}$: (Navigate A $\mathtt{w}_2$ $\mathtt{w}_3$), $a^{\mathtt{A}}_{wait}$: (Wait A), $a^{\mathtt{A}}_{wait}$: (Wait A)$\}$. The solution $\Pi^{\mathtt{A}}$ navigates rover A to the location $\mathtt{w}_3$ and waits two execution cycles until the plan execution of rover B, $\Pi^{\mathtt{B}}=\{a^{\mathtt{B}}_{7}$: (Analyze B r $\mathtt{w}_3$), $a^{\mathtt{B}}_{8}$: (Comm-rover B

A r w_3), a_9^{B} : (Navigate B w_3 w_2)}, which communicates the results of the analysis of the rock r to rover A, finishes.

5.5 Conclusions

In this chapter we have presented a multi-agent reactive planning and execution model that enables agents to recover from failures through collaboration with other agents. The model is embedded into a planning and execution system where an execution agent receives a plan from a planning agent, and the mission of the execution agent is to monitor, execute, and repair the given plan when a failure occurs.

We augmented the capabilities of the *REACTIVE PLANNER* (RP) with a Collaborative Repair behavior that allows execution agents to solve plan failures collectively by finding an agent that can help out to restore the execution plan of the failing agent. If so, the two agents work together to reach their goals. The principal objective of the multi-agent reactive planning and execution model is to ensure the continuous and uninterruptedly flow of the execution agents.

We used the multi-agent version of the motivating example introduced in Chapter 3.2, the Planetary Mars scenario, to conduct the explanation of the Collaborative Repair process. We also formalized the essential aspects of our Collaborative Repair process on the basis of independent agents that communicate to each other via the publicization of services and the construction of merged plan structures.

Chapter 6

Experimental evaluation in a multi-agent environment

"The achievements of an organization are the results of the combined effort of each individual."

(Vincent Lombardi)

This chapter presents the experimental evaluation of our MARPE model for solving plan failures. We conducted several tests to evaluate the reactiveness and performance of the MARPE approach along a complete execution of different planning tasks in various planning domains. In the experimentation carried out, we also compare the core of the MARPE model with other repair configurations, including an individual plan repair approach and a deliberative planner that finds a joint solution with a centralized planning mechanism.

In the following, we present the setup of the various repair configurations we used in the experiments, we introduce the domains and the specification of the multi-agent planning tasks and then we show and analyze the obtained results. Finally, we conclude and outline some discussions.

6.1 Plan repair in a multi-agent context

As we exposed in the previous chapter, the MARPE model aims to promptly and reactively solve a plan failure of an agent at execution time. The failing agent requests for help from one or more agents, but the repair process intends to be done with a maximum of two agents, such as we explained in Section 5.4.2 of Chapter 5, the requester agent and the helper agent. Agents work in a shared environment, and each agent is provided an individual plan that has to execute along with the plans of the rest of agents in the environment. Plans are independent of each other, but they all need to be successfully executed in order to solve the overall planning task. When an agent fails its mission of executing its plan, it demands collaboration to the rest of agents with the aim of applying a quick repair that allows it to resume its plan execution. Likewise, the helper agent that assists the failing agent attempts to bring the least possible distortion to its plan. In other words, the actual intention of the MARPE model is to fix a situation rapidly and allow the two execution agents, the requester agent and the helper agent, to resume their plans execution. A different choice to fix a failure is to start from scratch by resorting to a centralized planning approach that computes a new plan for each agent in the environment under the new current situation.

We are thus interested in comparing these two collaborative opposed approaches and validate the effectiveness of the MARPE model versus a deliberative and centralized approach that takes into account the plans of all the agents at the time of repairing a failure. For this purpose, we designed several multi-agent repair configurations that are explained in detailed in Section 6.1.2. Previously, we analyze the adaptation of the single-agent repair mechanims evaluated in Section 4.2.3 of Chapter 4 to a multi-agent context.

6.1.1 Adapting single-agent repair methods to a multi-agent context

In the single-agent experiments presented in Chapter 4, we tested three repair methods: our reactive planner (the core of the RPE model), and two deliberative methods, the LPG-Adapt mechanism and a replanning approach with the LAMA planner. The results showed that replanning was slightly more costly than using plan adaptation but, in contrast, LAMA was able to find much shorter plans than LPG-Adapt (see Table 4.8).

During a failure, the plan of an agent can suffer delays when the recovery mechanism changes the plan to solve the failure. The situation grows up in a multi-agent environment, where an agent can not only delay its plan because of a self-repairing process but either for collaborating to repair the failure of another agent. That is, a fault can affect the plan execution of one or two agents. Hence, in a multi-agent context, we aim to minimize the delays in the plans of the agents, and thus in the overall plan *makespan*, when a change needs to be applied in the plans to solve a failure. More specifically, our ultimate objective is to highlight the benefits and limitations of using our MARPE model, a multi-agent reactive framework that employs a collaborative plan repair technique for solving failures in planning applications. Specifically, we want to highlight the behavior of a multi-agent and reactive planning approach. Therefore, we are interested in evaluating two features:

- **collaborative repair versus individual plan repair**: a comparison of the behaviour of MARPE against an individual plan repair carried out by the failing agent. We want to check whether a collective plan repair turns out to be more beneficial than having agents independently repairing their plans.
- **reactive versus deliberative planning**: a comparison of MARPE against a deliberative planning approach. We want to assess the performance and ability of fixing failures of our reactive planning approach (incorpored into the MARPE model) in comparison to a deliberative procedure [1] (replanning). We must

[1]We use the LAMA planner, but results can be easily reproduced by simply replacing the LAMA

note that, as we mentioned in Section 5.4 of Chapter 5, a MARPE solution is only formed with the two agents that are directly involved in the failure solution whereas replanning is a centralized approach in which all agents give up their current plans in order to solve the failure, which may imply re-allocating goals to some agents.

The reactive planner introduced in Chapter 3 is a single-agent (S) reactive (R) technique for individual agents. Our multi-agent reactive planner is an extension of the *Single-Reactive* (SR) planner to a multi-agent context. Thereby, the framework presented in Chapter 5 is a multi-agent (M) reactive (R) planning technique that attempts to solve a failure collaboratively. We also identify a single-agent (S) deliberative (D) method when the failing execution agent calls its associated planner which computes a plan to solve the failure by its own. Finally, the multi-agent (M) deliberative (D) replanning approach can be viewed as an extension of the *Single-Deliberative* (SD) method.

In summary, we distinguish four different repair methods. We show the acronym of the repair method along with its description based on the characteristics explained in this section.

- SR: **S**ingle **R**eactive planning method; the model presented in Chapter 3.
- SD: **S**ingle **D**eliberative method; a plan is calculated by the planning agent associated to the execution agent.
- *Multi-**R**eactive* (MR): **M**ulti **R**eactive method; our MARPE model.
- *Multi-Deliberative* (MD): **M**ulti **D**eliberative method; the replanning approximation.

In a multi-agent system where all agents work in the same environment, a solution obtained with SR, SD, or MR could interfere with the actions of the plan

planner by LPG-Adapt

window of any other agent, in which case one of the two agents will detect the new failure and will call the corresponding repair method. As it is explained in Chapter 5, the plan calculated with a repair method (SR or MR) is a sequence of actions that repairs the current plan window, as opposite to a deliberative method (SD or MD) which synthesize a complete plan from the current state to the goals. We recall that the final objective of the agents is to have their plans executed so as to achieve the desired goals.

6.1.2 Repair configurations

In this section, we design three different combined repair configurations, each defined as a combination of two out of the four methods (SR, SD, MR or MD) exposed in the previous section. Table 6.1 shows the composition of the three different repair configurations, which we will call *individual repair*, *reactive repair* and *deliberative repair*. As we can observe, we use the SR method as a basis in all the configurations. This is because this repair method, as it was detailed in Chapter 4, turns out to be the quickest and most effective repairing mechanism to fix a single plan failure. Therefore, a call to the SR repair method is always made at first place in all the configurations in order to find a quick solution, if possible; otherwise, we use the repair methods that actually give name to the configuration.

Table 6.1: Repair configurations designed as a combination of two repair methods.

individual repair	reactive repair	deliberative repair
SR + SD	SR + MR	SR + MD

6.1.2.1 Individual repair configuration

This configuration is aimed at evaluating an individual response of the failing agent with no collaboration from any other agent.

The individual repair configuration consists in applying the following repair methods, in this order:

1. First, the agent applies the SR method as exposed in Chapter 3.
2. In case the agent is not capable of reactively self-repairing its plan, it resorts to the SD repair method, requesting a plan to the deliberative planner comprised in its associated planning agent.

6.1.2.2 Reactive repair configuration (MARPE model)

In this configuration, we stress the use of a reactive solution. Thereby, this configuration represents the essence of our MARPE model:

1. First, the agent individually applies the SR planning method as in the individual configuration. If the self-repaired solution of the agent does not succeed, the configuration then draws upon a multi-repair solution.
2. The MR repair method searches for a multi-reactive solution where a potential helper agent agrees to collaborate in order to solve the failure. Unlike the SR method, the agent identified as helper in MR has an active role working together with the requester agent towards a solution.

One additional observation must be recalled out here. If an agent is working together with another agent and a failure occurs in its plan, the agent directly works as it was explained in Sections 5.4.4.1 and 5.4.4.2 without calling the SR method at first place or the MR method from the beginning. That is so because the recovery methods offered in Sections 5.4.4.1 and 5.4.4.2 denote parts of our MARPE model. Hence, for simplicity during the current Chapter each time an agent forms a reactive solution and a failure occurs during its execution, we will say the agent activates the MR method.

6.1.2.3 Deliberative repair configuration (replanning)

In this configuration, in case the usual SR repair method does not succeed, all of the agents actively participate in finding a deliberative solution. When the MD method is invoked, all execution agents interrupt their plan simulation and they communicate their current state to a centralized deliberative planner, which is responsible for finding a global solution. Whereas the reactive approach only involves the agents that are potential helpers for the failure at hand, in the deliberative configuration, all the agents give up their plans and they collaboratively call a centralized deliberative planner.

Table 6.2 summarizes the type and number of agents involved in each repair method as well as the number of agents whose plans participate in such repair method.

Table 6.2: Overview of the type and number of agents in each repair method.

	execution agents	planning agents	modified plans
SR	1	-	1
SD	1	1	1
MR	m	-	m
MD	n	1	n

Let's assume n is the total number of execution agents involved in a particular repair. The SR and SD methods involve one execution agent or its associated planning agent, in the case of the SD method, and they only modify the plan of the involved execution agent. In the MR method, $m = 2$ execution agents are involved in the reactive solution, and their respective m plans will be likely modified. On the other hand, only a centralized planning agent participates in the MD configuration but working with the plans of all the n execution agents in the system. The SR, SD and MR methods can generate solutions with disagreement with the plans of the other $n-1$ (SR and SD methods) or $n-m$ (MR method) execution agents, in which case during the execution the failure will be detected and solved with the repair methods.

6.2 Multi-agent planning tasks

There exist three type of agents in the MARPE approach that interact during the execution of a *Multi-Agent Planning* (MAP) task, namely:

1. execution agents, as specified in Chapter 5, which include all the executive machinery and the reactive planner
2. planning agents associated to the execution agent and each one comprises a deliberative planner
3. the simulator agent.

Unlike the single-agent experiments, which were only aimed at comparing the performance of the repair mechanisms, in the multi-agent context we execute the planning tasks and, therefore, the simulator also intervenes in the planning and execution system. Execution agents, the principal entities of the MARPE model, receive a domain and a problem file, which encode their view of the MAP task, including the planning actions that model their capabilities and the goals they must achieve. Execution agents then perceive the initial state via the simulator, which sends each execution agent the set of fluents known to the agent; i.e., the fluents that affect the preconditions and effects of their actions. Once the planning tasks of the agents are defined (actions, goals and initial state), each execution agent calls its deliberative planner to obtain a plan that solves the task and they continue with the planning and execution process.

It is important to highlight that if the simulator detects that two actions are conflictive, the simulator only executes one of the two actions.

As in the single-agent experiments, execution agents receive the task files encoded in PDDL3.1 whereas the planning agents and simulator receive a task written in PDDL2.1. As we explained in Section 2.1.2.3 of Chapter 2, the simulator of PlanInteraction was adapted from the original simulator of PELEA, which works with PDDL2.1. For the conversion from PDDL3.1 to PDDL2.1 and viceversa, we employ the converters provided in PELEA.

6.2.1 Domain agentization

We generated MAP tasks from three planning domains, namely the `rovers` domain, the `elevators` domain and a `transport` domain. Our objective is to define MAP tasks such that the task goals are distributed across agents and each agent is then responsible and capable of achieving the goals specified in its individual planning task. That is, in the absence of unexpected events that would prevent agents from executing their plans, agents will successfully execute their plans and achieve their assigned goals. Thus, our purpose is to have a group of independent agents executing their individual plans in a common environment.

For generating the MAP tasks, we adapted the corresponding single-agent domains from the *INTERNATIONAL PLANNING COMPETITION* (IPC) problem suites by using the agentization that was firstly presented in [110] [2]. This same agentization was later used in the factored representation of the *Multi-Agent Planning Domain Definition Language* (MA-PDDL) of the distributed track at the first *Competition of Distributed and Multi agent Planners* (CoDMAP) [2]. Actually, the CoDMAP celebrated at the *International Conference on Automated Planning and Scheduling* (ICAPS) of 2015 has contributed towards establishing the foundations for a standard representation of MAP tasks.

In the following sections, we explain the agentization of the three domains.

[2]Domains are also available at http://users.dsic.upv.es/grupos/grps/tools/map/fmap.html excepts for the `transport` domain, which is inspired from the `driverlog` domain

6.2.2 Rovers domain.

The original formulation of the `rovers` domain was presented in the IPC of 2002. This domain was already used in Chapter 4 and features rovers that have the necessary abilities to collect soil and rock samples or take images.

The agentization of the `rovers` domain consists of creating a planning task per rover. All the agents are of the same type, and they will typically have the same set of planning capabilities (actions), but there are some exceptions as for the rovers that are equipped with a camera, which are the only ones that can take images. The single-agent specification [3] (see Appendix B.1) already includes a `rover` parameter in all the actions. However, this `rover` parameter is not specified in the **:types** section as an `agent` type as it is required in our multi-agent modeling (see Appendix B.2.1). Additionally, in order to promote a higher degree of cooperation between agents, we define an extra delivering capability which will be explained in Section 6.4.1.

6.2.3 Elevators domain.

The `elevators` domain was used as first time in the IPC of 2008. The domain simulates a real-world problem where there is a building with N floors, and K elevators that stop at each floor. There are several passengers, for which their current location (i.e. the floor they are) and their destination are given. The planning task is to find a plan that moves the passengers to their destination floor.

We adapted the `elevators` domain presented in the IPC to make floors be accessible by all elevators. We thus define planning tasks where elevators do not have to stop at intermediate floors for that the passengers board another elevator that takes them to their destination.

The agentization of the `elevators` domain consists of creating a planning task per elevator. Like the `rovers` domain, all the agents are of the same type, and

[3] This is the same specification from the original one of the IPC, but with some new capabilities

they will have the same set of actions. As we mentioned before, our single-agent specification (see Appendix C.1) is not the same as the original specification of the IPC. We remove the capacity conditions of the `elevator` to generate consistent failures, and we also remove the restricted floor access that some elevators have to avoid dependencies between agents and promote more cooperation during the repair of a plan failure. The single-agent specification already includes an `elevator` parameter in all the actions. We model the `elevator` parameter in the **:types** section as an `agent` type as it is required in our MARPE model (see Appendix C.2.1). More details of this domain will be explained in Section 6.4.2.

6.2.4 Transport domain.

The `transport` domain is a modified version of the `driverlog` domain presented in the IPC of 2002. This domain is an important problem in the fields of transportation and distribution [51]. In our `transport` domain, we removed the drivers and all its abilities. Thus, we only keep a fleet of trucks that can drive between locations. The trucks can load or unload packages, and the objective is to transport packages between locations, ending up with a subset of the packages, and the trucks at specified destinations.

The agentization of the `transport` domain consists of creating a planning task per trucks. Like the `rovers` and `elevators` domain, all the agents are of the same type, and they will have the same set of actions. The single-agent specification (see Appendix D.1) was modeled to already include a `truck` agent parameter in all the actions. Obviously, we specified the `truck` in the **:types** section as an agent type as it is required in our model. For the single-agent specification, we keep the domain with three actions: drive the `truck` from one location to another, load a package from one location to a `truck` and unload a package from a `truck` to a specific location. Additionally, and to promote a higher cooperation between agents, we modeled two extra capabilities in the multi-agent specification (see Appendix D.2.1), which will be explained in Section 6.4.3.

6.3 General overview of the experiments

In this section, we explain the common elements of the experiments we carried out in all the domains. We discuss the parameters that we measure and the meaning of the data in the results tables. The particular analysis of the outcomes obtained for the three domains (see Section 6.4) is separately explained in the section corresponding to each domain. Specifically, we designed two scenarios:

a. **one-time failure scenario**: we trigger one or several failures that occur at a time in the initial agents' plans.

b. **two-times failure scenario**: we trigger two consecutive failures; the first failure is the same as in the one-time failure scenario and then a second failure is induced on the repaired plan after the first failure. Since the plan calculated by each repair configuration can be different after the first repair, the second failure may be triggered at different times on the repaired plans.

It is important to highlight that each triggered failure may affect one or several agents. We also note that failures other than the triggered ones may happen in the two scenarios as a consequence of the resulting plan repair and the behaviour of the repair approach. We will call this type of failure *collateral* failures.

We selected six MAP tasks for each domain where each task corresponds to a problem from the IPC. The MARPE model is implemented in Java, and we run all tests on a GNU/Linux Debian PC with an Intel 7 Core i7-3770 CPU @ 3.40GHz x 8, and 8 GB RAM.

6.3.1 Structure of the results tables

In this section, we explain the structure of the results tables and the parameters to be measured. All the results tables for the three domains are exactly the same so we will refer to Table 6.5 (one-time scenario) and Table 6.7 (two-times scenario) of the `rovers` domain when explaining the general structure of the tables.

A row of a table represents a failure applied to a MAP task, and the columns denote:

1. the column **task** denotes the particular MAP task in which the failure is applied, and the number of agents and goals of the task.
2. the column **fail** indicates the induced failure (the meaning of the failure is explained in the sections that show the obtained results). A failure in the one-time failure scenario is denoted as Fi, meaning that we triggered a single failure; and, in the two-times failure scenario is denoted as Fi-Fj, meaning that we trigger two consecutive failures, Fi and then Fj.
3. the columns of the repair methods show the outcome of applying each method in the three repair configurations: *individual repair*, *reactive repair* (our MARPE model), and *deliberative repair*, which work as follows:
   - the individual repair configuration activates first SR; if SR cannot solve the failure, the SD method is then activated to repair the plan failure.
   - the reactive repair configuration calls the SR method if the agent is not working together with another agent, or the MR method if the agent is collaborating; if the SR is not able to repair the failure, then the MR method is activated.
   - the deliberative repair configuration activates first SR; if SR cannot solve the failure, the MD method is then invoked.

The MAP tasks of the agents are simulated until their completion and we measure the following parameters:

i. the number of reached goals at the end of the simulated plan execution compared to the number of goals in the initial MAP task. This is represented by the column **reached goals** as can be seen in Table 6.5.

ii. the overall number of execution cycles of the complete simulation of the MAP task. This parameter is represented in the column **cycles** of all results tables, as can be observed in Table 6.5.

iii. the gap between the time in which the goals of the agent are reached and the time where are expected to be reached. A positive number is when the agent finishes later and generally, but not always, the goals of the agent are reached. A negative number means the agent finishes earlier, in which case the goals of the agent usually are not reached. A zero number means the agent finishes on time and the goals of the agent are reached.

iv. the computation time of the MAP task execution. This value is represented under the column **real time** in the results tables (see Table 6.5 as an example).

6.3.2 Figures of the results tables.

In this section, we explain the meaning of the figures that appear in the results tables. For each repair configuration, we show the total number of times the repair methods (SR, SD, MR, or MD) are activated versus the number of times the method is capable of solving the failure. More specifically:

1. **Meaning of x/y.** The notation x/y of the results tables show that y is the number of times the method is invoked and x the number of times the method successfully solves the failure.

For instance, in failure F1 of task 1 (see the one-time failure scenario of Table 6.5), the value of SR is $x/y = 1/1$, meaning that SR is activated once and finds a solution for such failure. Consequently, the SD method is not invoked since SR was capable of solving the failure. The same value $x/y = 1/1$ appears in the SR of the reactive and deliberative configurations, which also indicates that these two methods are able to solve the failure. Likewise, the rest of methods are never invoked.

In the F1-F2 failure of task 1 in the two-times failure scenario presented in Table 6.7, the value of SR in the reactive and deliberative configurations is $x/y = 2/2$, meaning that SR is activated twice and finds a solution in the two cases. In this case, the two activations of the SR method correspond to the two consecutive failures, F1 and F2.

2. **One same failure may activate two or more repair methods.** If a method is not capable of solving a failure, the next method in the configuration is activated. Then, the y value of two different methods in the same configuration may comprise a call for the same failure.

   For instance, in failure F9 of task 5, the value of SR in the individual configuration is $x/y = 0/1$, meaning that SR is activated once but is not capable of finding a solution. Next, the SD of the individual configuration is activated but is not able to find the solution either ($0/1$). Consequently, `agent2` finishes its plan execution sooner (see the delay of `agent2`), and only 4 out of 6 goals are reached. The same situation occurs in the SR method of the reactive configuration, which does not repair the failure, and the MR is consequently activated. Likewise, MR does not attain repairing the task ($0/1$). In contrast, the MD method of the deliberative configuration repairs the plan failure ($x/y = 1/1$).

3. **Failures for multiple agents.** In both scenarios, the number of times a method is activated, y, may refer to a failure produced in a single agent or a failure produced in multiple agents.

For instance, in the one-time failure scenario of Table 6.5, the value of SR for F14 of task 1 is $x/y = 2/2$, which means that SR is activated twice and finds a solution in both cases. In this case, the two activations of SR stem from the fact that F14 provokes a failure that affects two agents, `agent1` and `agent2`. The value of SR in the three repair configurations stand for the same double failure.

In the two-times failure scenario, identifying a multiple agent failure is not straightforward because the value x/y not only shows the result of the first failure, which may affect one or several agents, but also the second consecutive failure, which may also affect several agents. Therefore, we identify cases where:

- $y = 2$, in which case each consecutive failure only involves a single agent
- $y > 2$, in which case either failure can affect more than one agent

For instance, in the two-times failure scenario of Table 6.7, the value of SR for F21-F22 of task 6 in the reactive and deliberative configurations is $x/y = 5/5$. The first failure F21 modifies the state of two agents, finding a solution for both agents; and the second failure, F22, affects three agents and finds a solution plan for all of them as well.

4. **Collateral failures.** The number of times a method is activated, y, may respond to the number of calls for solving the original failure or for solving a collateral failure. Thus, a failure repaired with the SR and MR methods may sometimes produce a collateral failure.

   For instance, in the one-time failure scenario of Table 6.5, the value $x/y = 2/2$ of SR for failure F2 in tasks 1 and 5 indicates that two failures are produced and repaired; the first one is the original F2 and the second one is a collateral failure.

Another example can be seen with F16 of task 5 in Table 6.5, where three of the four failures ($y = 4$) are the failures that F16 provokes in three agents. Each agent repairs its plan, which finishes two cycles later over the original plan (see column delay of the three agents). Notice, `agent3` delays its plan execution in one more cycle ($delay = 2$) than the other two agents ($delay = 1$). The reason is because the solution of `agent3` generates a collateral failure, which `agent3` repairs activating its SR in the three configurations.

The MR method can also produce a collateral failure, like in F11 of task 1 in the one-time failure scenario (see Table 6.5), where the MR of the reactive configuration repairs the failure forming a new multi-reactive solution and, when the agents finishes the execution of the multi-reactive solution, a collateral failure is produced in the plan of the helper agent `agent1`, which is repaired with the SR method; i.e., the SR method is activated twice; one for attempting to solve F11, which turns out not to be a successful repair, and the second one to repair a collateral failure, which is successfully solved; i.e. $x/y = 1/2$. The same occurs in F13 of task 1.

In failure F1-F13 of task 4 in the two-times scenario (see Table 6.7), the value of SR in the reactive configuration is $x/y = 2/3$, meaning that SR is activated thrice and finds a solution in two of the three cases. The first one is to successfully solve F1; the second one is to unfruitfully solve F13; and the third one is for successfully solving a collateral failure detected when agents finishes the execution of the multi-reactive solution formed by the MR to repair the second failure. In contrast, the MD solution of the deliberative configuration does not generate a collateral failure.

5. **Parameters.** Same values of x/y in different repair methods may sometimes render a different number of cycles, and values of real time in the failures of the one-time and two-times failure scenarios.

   For instance, in the one-time failure scenario of Table 6.5, the value of SR in

F18 of task 6 ($x/y = 2/2$) means that SR is activated twice and finds a solution in the two cases. The first failure refers to the original failure F18 and the second one to a collateral failure. As we can observe, the value of SR in the three repair configurations is the same. However, we can see some differences in the number of cycles and real time execution. The deliberative configuration takes one more cycle than the individual and reactive configurations. This situation is normal because the deliberative repair requires some extra time to calculate the first plan for the agents; i.e., the central planner of MD needs to wait until all executing have sent their knowledge of the initial state in order to compound a global vision of the world state to generate the plan, and once the plan is calculated, it assigns the actions to each execution agent. The same happens in the two-times failure scenario with the failure F3-F2 of task 3 (see Table 6.7).

Once explained the general meaning of the data for the results tables, in the following we present the particular analysis of the experimental results for each domain.

6.4 Experimental results

In this section, we present the obtained results for the three domains. First we show the results for the `rovers` domain with the one-time and two-times scenarios. Next, we present the results for the `elevators` and `transport` domains with the one-time scenario. Finally, in the next section we outline some conclusions and discussions of this chapter.

6.4.1 `rovers` domain

The multi-agent `rovers` domain presents some differences in relation to the single-agent version. In addition to the new capabilities introduced in Chapter 4, we endow `rover` agents with rover-to-rover communication abilities represented by the operators (Communicate_rover `?r1` - rover `?r2` - rover `?s` - sample `?w1` - waypoint `?w2` - waypoint), or (Communicate_image_rover `?r1` - rover `?r2` - rover `?o` - objective `?m` - mode `?w` - waypoint) that allows a `rover` `?r` to communicate a rover `?r2` the results of the samples analysis or the taken image at waypoint `?w`. This new ability is included to promote an alternative joint way of repairing failures (see Appendix B.2.1).

In this domain, a `rover` agent collaborates in the multi-repair solution employing a total number of five capabilities: seek for more samples (new capability added in the Chapter 4), analyze rock samples, analyze soil samples, take images, and communicate any previous results to any other `rover`. We designed two different capability configurations, R1 and R2, for the `rovers` domain (see Table 6.3).

Table 6.3: Two configurations of capabilities for the `rovers` domain

conf	description
R1	rovers have the abilities to seek for more samples, analyze rock and soil samples, and take image; but they do not have the rover-to-rover communication skill to communicate results between them.
R2	rovers have all the abilities (seek for more samples, analyze rock and soil samples, take image, and communication rover-to-rover).

Configurations R1 and R2 were executed with the *one-time failure* scenario explained above. This test allows us to discuss the performance of the three repair configurations under the same failing situations: when agents are not able to communicate results to each other (R1), or when they have the capability of rover-to-rover communication (R2). We will see in the results that adding more capabilities to the rovers also increases the reactive solutions. On the other hand, we executed the *two-times failure* scenario only with the R2 `rover` configuration because since the plans calculated for the two `rover` configurations, R1 and R2, will be different after the first repair, applying the same second failure in R1 and R2 may be impossible. Thereby, it is pointless to compare both configurations when the two consecutive failures are not the same. Additionally, configuration R2 is more interesting for us because it gives rise to more multi-reactive solutions. This is ultimately our aim in order to test the benefits of our MARPE model.

Table 6.4: Failures generated for the `rovers` domain

	failure	description
single failures	F1	one `rover` loses the good maps to navigate from one to other waypoint.
	F2	the objective of one `rover` is not visible from the waypoint.
	F3	one rover's camera loses calibration.
	F4	one `rover` loses the results of sample analysis.
	F5	one `rover` loses the results of the image.
	F6	one `rover` loses the capability to take image.
	F7	one `rover` loses the capability to seek for more samples.
	F8	one `rover` loses the capability to analyze samples (both soils and rocks).
	F9	one `rover` loses the capability to analyze soils or rocks (just one of both).
	F10	one `rover` loses the sample when it is about to analyze in a waypoint.
compound failures	F11	F10 and F7 (same rover).
	F12	F10 and F8 (same rover).
	F13	F4 and F7 (same rover).
	F14	F3 and F4 (different rovers).
	F15	F5 and F6 (same rover).
	F16	2xF1 and F2 (different rovers).
	F17	F3 and F6 (same rover).
	F18	F3 and F2 (same rover).
	F19	F10 and F1 (different rovers).
	F20	F4 and F8 (same rover).
	F21	F1 and F18 (different rovers).
	F22	F10 and F19 (different rovers).
	F23	F4 and F7 (different rovers).

Table 6.4 shows all the failures we generated for the `rovers` domain, which we divided into single and compound failures. A single failure alters a fluent in the current world state like failure F3, in which the camera loses calibration. Compound failures change two or more fluents in the current world state, such as failures from F11 to F23. Notice that a compound failure may affect not only one agent (F11-F13), but two or more agents (F14 or F16). We randomly chose and applied a single or compound failure from Table 6.4 during the simulated plan execution. The rovers' actions for solving the failures can be any combination of this set of actions:

1. the `rover` seeks for more soil samples
2. the `rover` seeks for more rock samples
3. the `rover` analyzes the samples.
4. the `rover` calibrates the camera.
5. the `rover` takes the image.
6. the `rover` communicates the samples analysis to the lander.
7. the `rover` communicates the taken image to the lander.
8. the `rover` communicates the samples analysis to another rover.
9. the `rover` communicates the taken image to another rover.

For instance, assuming that we apply the single failure F10, the solution plan of the `rover` is to seek for more soil or rock samples at the specific waypoint, analyze them and communicate the samples analysis to the lander. If we apply the single failure F3, in which the camera loses calibration, the recovery solution requires to calibrate the rover's camera again.

In the case of a compound failure like F12, in which the failing `rover` loses the sample and also the capability to analyze samples, the solution plan of the `rover` requires that another `rover` analyzes the samples and communicates the results to the lander (in configuration R1) or to the failing `rover` (in configuration R2), which in turn will transmit the results to the lander.

The six tasks we executed of the `rovers` domain correspond to the following problems:

- our task 1 has two rovers and four goals
- our task 2 has two rovers and five goals
- our task 3 has two rovers and three goals
- our task 4 has two rovers and six goals
- our task 5 has three rovers and six goals
- our task 6 four rovers and eight goals

6.4.1.1 one-time failure scenario

Table 6.5 shows the results for the one-time failure scenario with the R1 `rover` configuration. As we can see, the deliberative repair configuration was the only one capable of achieving the goals of all the tasks (compare columns task and goals reached). This is reasonable because the *Centralize Planner* (CP) of the MD repairs the plan failures with a global vision of the world state. For instance, in F6 of task 1, `rover1` loses the ability to take an image and activates the SR of the reactive repair. SR is not capable of repairing the plan failure, and MR is activated to request another agent to take the image and to communicate the results to the lander; unlike to R2 configuration, where the `rovers` may use the rover-to-rover communication skill to communicate the sample analysis to the failing `rover`. The MR is also not able to repair the plan failure. In contrast, with the deliberative repair the CP of the MD generates a new plan where `rover2` executes all the work of `rover1`.

Table 6.5: Results for the one-time failure scenario with the R1 `rovers` configuration

		individual repair										reactive repair										deliberative repair									
		RP										RP										RP									
task	fail	SR	SD	goals reached	delay				cycles	real time		SR	MR	goals reached	delay				cycles	real time		SR	MD	goals reached	delay				cycles	real time
					(agents)					1:8 seg					(agents)					1:8 seg					(agents)					1:8 seg
					1	2	3	4							1	2	3	4							1	2	3	4		
	F1	1/1	0/0	4	0	1	-	-	8	67.72		1/1	0/0	4	0	1	-	-	8	67.76		1/1	0/0	4	0	1	-	-	8	67.76
	F2	2/2	0/0	4	2	0	-	-	8	66.74		2/2	0/0	4	2	0	-	-	8	66.57		2/2	0/0	4	2	0	-	-	9	75.00
1	F14	2/2	0/0	4	1	4	-	-	11	90.53		2/2	0/0	4	1	4	-	-	11	90.53		2/2	0/0	4	1	4	-	-	11	90.73
4 goals	F11	0/1	0/1	2	-4	0	-	-	7	58.51		1/2	1/1	4	6	3	-	-	12	98.59		0/1	1/1	4	3	3	-	-	10	82.59
2 agents	F12	0/1	0/1	2	-4	0	-	-	7	58.33		0/1	1/1	4	3	4	-	-	11	90.60		0/1	1/1	4	5	5	-	-	12	98.51
	F13	0/1	0/1	2	-3	0	-	-	7	58.52		1/2	1/1	4	5	5	-	-	12	98.33		0/1	1/1	4	5	3	-	-	12	98.52
	F6	0/1	0/1	3	-2	0	-	-	7	58.45		0/1	0/1	3	-2	0	-	-	7	58.39		0/1	1/1	4	2	4	-	-	11	90.56
	F11	0/1	0/1	3	-5	0	-	-	9	74.59		1/2	1/1	5	5	3	-	-	13	106.75		0/1	1/1	5	8	6	-	-	17	138.70
	F12	0/1	0/1	3	-5	0	-	-	9	74.55		0/1	0/1	3	-5	0	-	-	9	74.45		0/1	1/1	5	7	6	-	-	16	130.53
2	F4	0/1	1/1	5	5	0	-	-	13	106.45		0/1	1/1	5	7	5	-	-	15	122.48		0/1	1/1	5	6	5	-	-	15	122.45
5 goals	F6	0/1	0/1	3	-4	0	-	-	9	74.43		0/1	0/1	3	-4	0	-	-	9	74.40		0/1	1/1	5	2	4	-	-	13	106.37
2 agents	F3	1/1	0/0	5	1	0	-	-	9	74.43		1/1	0/0	5	1	0	-	-	9	74.42		1/1	0/0	5	1	0	-	-	10	82.45
	F2	1/1	0/0	5	2	0	-	-	10	82.54		1/1	0/0	5	2	0	-	-	10	82.40		1/1	0/0	5	2	0	-	-	11	90.43
	F1	1/1	0/0	5	2	0	-	-	10	82.48		1/1	0/0	5	2	0	-	-	10	82.33		1/1	0/0	5	2	0	-	-	11	90.42
	F23	0/1	1/1	5	5	0	-	-	13	107.69		0/1	0/1	3	-2	0	-	-	6	63.77		0/1	1/1	5	4	5	-	-	14	115.52
	F1	1/1	0/0	5	1	0	-	-	9	74.82		1/1	0/0	5	1	0	-	-	9	74.93		1/1	0/0	5	1	0	-	-	10	82.86
	F11	0/1	0/1	3	-5	0	-	-	9	74.61		1/2	1/1	5	5	3	-	-	13	106.55		0/1	1/1	5	7	7	-	-	16	130.51
3	F6	0/1	0/1	3	-4	0	-	-	9	74.47		0/1	0/1	3	-4	0	-	-	9	74.58		0/1	1/1	5	5	3	-	-	14	114.66
5 goals	F3	1/1	0/0	5	1	0	-	-	9	74.45		1/1	0/0	5	1	0	-	-	9	74.56		1/1	0/0	5	1	0	-	-	10	82.49
2 agents	F2	1/1	0/0	5	2	0	-	-	10	82.48		1/1	0/0	5	2	0	-	-	10	82.58		1/1	0/0	5	2	0	-	-	11	90.46
	F15	0/1	0/1	4	-1	0	-	-	9	74.44		0/1	1/1	5	3	3	-	-	12	98.50		0/1	1/1	5	3	4	-	-	13	106.55
	F10	1/1	0/0	6	1	0	-	-	10	82.52		1/1	0/0	6	1	0	-	-	10	82.60		1/1	0/0	6	1	0	-	-	10	82.89
	F1	1/1	0/0	6	1	0	-	-	10	82.54		1/1	0/0	6	1	0	-	-	10	82.53		1/1	0/0	6	1	0	-	-	10	82.62
4	F13	0/1	0/1	3	-5	0	-	-	9	74.46		1/2	1/1	6	4	5	-	-	14	114.50		0/1	1/1	6	6	3	-	-	15	122.45
6 goals	F11	0/1	0/1	4	-4	0	-	-	9	74.53		1/2	1/1	6	5	3	-	-	14	114.64		0/1	1/1	6	3	6	-	-	15	122.55
2 agents	F12	0/1	0/1	3	0	-7	-	-	9	74.61		0/1	0/1	3	0	-7	-	-	9	74.63		0/1	1/1	6	9	11	-	-	24	193.63
	F15	0/1	0/1	5	-1	0	-	-	9	74.54		0/1	1/1	6	3	3	-	-	12	98.60		0/1	1/1	6	2	4	-	-	13	108.12
	F6	0/1	0/1	5	0	0	-2	-	7	59.55		0/1	0/1	5	0	0	-2	-	7	59.67		0/1	1/1	6	2	3	2	-	10	85.38
	F12	0/1	0/1	4	0	-3	0	-	7	59.26		0/1	0/1	4	0	-3	0	-	7	59.19		0/1	1/1	6	6	4	4	-	13	107.84
5	F2	2/2	0/0	6	0	0	2	-	8	67.65		2/2	0/0	6	0	0	2	-	8	67.16		2/2	0/0	6	0	0	2	-	9	75.17
6 goals	F16	4/4	0/0	6	1	1	2	-	8	67.33		4/4	0/0	6	1	1	2	-	8	67.24		4/4	0/0	6	1	1	2	-	9	75.17
3 agents	F3	1/1	0/0	6	0	0	1	-	7	59.34		1/1	0/0	6	0	0	1	-	7	59.38		1/1	0/0	6	0	0	1	-	8	67.08
	F9	0/1	0/1	4	0	-3	0	-	7	59.11		0/1	0/1	4	0	-3	0	-	7	59.30		0/1	1/1	6	4	6	4	-	13	107.37
	F20	0/1	0/1	6	0	-2	0	0	7	60.21		1/2	1/1	8	5	4	0	0	12	101.07		0/1	1/1	8	4	4	6	3	13	109.78
	F21	3/3	0/0	8	1	0	3	0	9	76.23		3/3	0/0	8	1	0	3	0	9	76.13		3/3	0/0	8	1	0	3	0	10	84.59
6	F18	2/2	0/0	8	0	0	3	0	9	76.26		2/2	0/0	8	0	0	3	0	9	76.78		2/2	0/0	8	0	0	3	0	10	85.77
8 goals	F9	0/1	0/1	6	0	-3	0	0	7	60.42		0/1	0/1	6	0	-3	0	0	7	59.98		0/1	1/1	8	3	5	3	3	12	100.23
4 agents	F6	0/1	0/1	6	0	0	-4	0	7	59.92		0/1	0/1	6	0	0	-4	0	7	60.45		0/1	1/1	8	3	2	5	3	12	100.42
	F4	0/1	1/1	8	0	3	0	0	9	76.75		1/2	1/1	8	5	4	0	0	12	100.22		0/1	1/1	8	3	5	3	3	12	100.25

On the contrary, the reactive repair was not able to reach all the goals in 28% of the execution failures. However, the reactive repair achieves more goals in fewer cycles than the deliberative and the individual repairs, as in failure F11 of task 2; where the reactive repair reaches all the goals in 13 cycles of execution, the deliberative repair in 17 cycles, and the individual repair does not reach all the goals.

The three repair configurations reach all the goals in 48% of the execution failures because failures are solved with the SR mechanism, except in F4 and F23 of task 2 and F4 of task 6. In this failures, SR is not capable of repairing the plan failure, and the agent activates the SD, MR, or MD of the individual, reactive or deliberative repair configurations, respectively. Obviously, as the SR solves the failures, the real-time results will be almost the same in the three repair configurations except for the deliberative configuration where sometimes the MD loses one execution cycle generating and allocating the solution plan to each agent.

Figure 6.1 compares the real-time results when the reactive and deliberative configurations reach all the goals. We decide not to compare the individual configuration because it only reaches all the goals in 48% of the execution failures. As we can see, the Figure 6.1 confirms that the reactive repair achieves all the goals in fewer cycles than the deliberative configuration. In the cases where the differences are minimal, such as F1 and F10 of task 4, the failure is solved with our SR mechanism, which highlight the good performance of the self-repair mechanism. Consequently, as it is shown in Table 6.5, the reactive repair also presents fewer delay in the agents' plan execution, such as the failure F15 of task 3 where the agents in the reactive repair present a delay of 3 cycles against the 4 cycles of the deliberative configuration. Opposite cases are due to collateral failures, which delay the agents' plan execution. For instance, in F2 of task 1, the solution plan is that `rover1` navigates to another waypoint where the objective is visible to take the images. This solution plan produces a collateral failure due to the `rover1` is not any longer in the specific location where it was supposed to be in order to analyze the soil or rock samples. The SR of the individual configuration repairs the collateral failure navigating `rover1` to the specific waypoint. Notice that as expectedly, if SR is capable of solving a failure, so it is SR of the reactive and deliberative configurations.

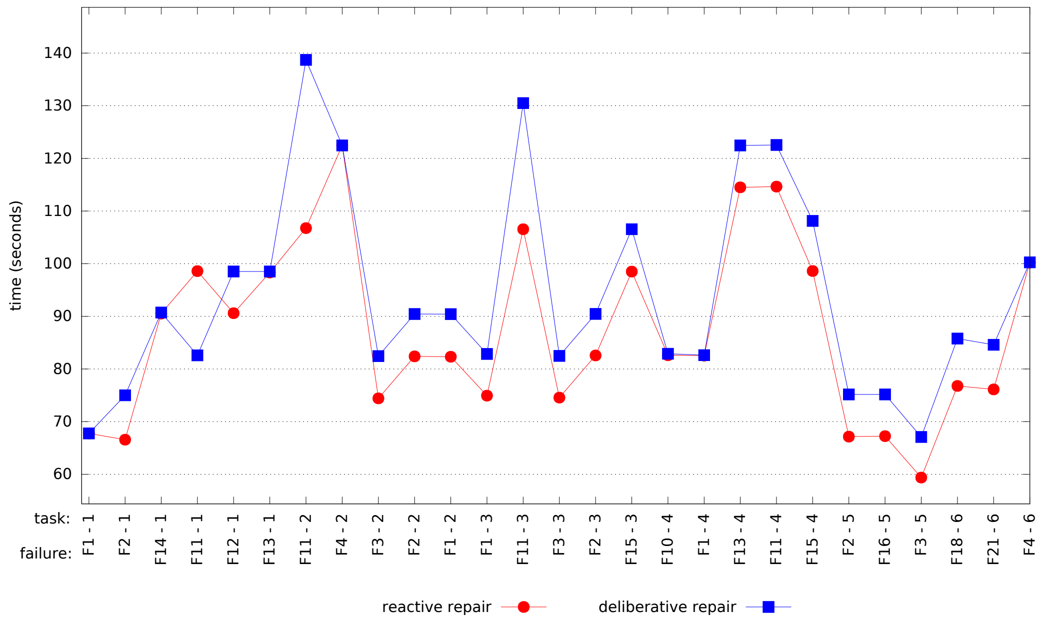

Figure 6.1: Real-time results when the reactive and deliberative repair configurations reach all the goals in the one-time scenario with R1 `rovers` configuration

Some compound failures that affects different agents are not detected by all the agents. For instance, the compound failure F23 of task 2 is applied over different agents; i.e., `rover1` loses the results of sample analysis and `rover2` loses the capability to seek for more samples. But only `rover1` detects the plan failure. `rover2` does not detect the failure because it does not affect its current plan execution. Thus, `rover1` activates its SR, which was not able to repair the plan failure and then the MR is activated, but MR was also not able to repair the failure because `rover2` has not the ability to seek for more samples. In contrast, the MD of the deliberative repair generates a solution plan where `rover1` seeks for more samples.

Table 6.6 shows the results for the one-time failure scenario with the R2 `rover`

Table 6.6: Results for the one-time failure scenario with the R2 `rovers` configuration

		individual repair									reactive repair									deliberative repair								
		RP									RP									RP								
task	fail	SR	SD	goals reached	delay				cycles	real time	SR	MR	goals reached	delay				cycles	real time	SR	MD	goals reached	delay				cycles	real time
					(agents)					1:8 seg				(agents)					1:8 seg				(agents)					1:8 seg
					1	2	3	4						1	2	3	4						1	2	3	4		
	F1	1/1	0/0	4	0	1	-	-	8	67.92	1/1	0/0	4	0	1	-	-	8	67.59	1/1	0/0	4	0	1	-	-	8	67.99
	F2	1/1	0/0	4	2	0	-	-	8	66.72	1/1	0/0	4	2	0	-	-	8	66.52	1/1	0/0	4	1	0	-	-	8	66.63
1	F14	2/2	0/0	4	1	4	-	-	11	90.65	2/2	0/0	4	1	4	-	-	11	90.67	2/2	0/0	4	0	4	-	-	11	90.56
4 goals	F11	0/1	0/1	2	-4	0	-	-	7	58.60	1/2	1/1	4	6	3	-	-	12	98.58	0/1	1/1	4	4	4	-	-	11	90.59
2 agents	F12	0/1	0/1	2	-4	0	-	-	7	58.56	0/1	1/1	4	4	4	-	-	11	90.41	0/1	1/1	4	5	6	-	-	13	106.59
	F13	0/1	0/1	2	-3	0	-	-	7	58.43	1/2	1/1	4	6	5	-	-	12	98.48	0/1	1/1	4	5	3	-	-	12	98.50
	F6	0/1	0/1	3	-2	0	-	-	7	58.64	0/1	1/1	4	3	3	-	-	10	82.32	0/1	1/1	4	2	4	-	-	11	90.57
	F11	0/1	0/1	3	-5	0	-	-	9	74.82	1/2	1/1	5	5	3	-	-	13	106.38	0/1	1/1	5	8	6	-	-	17	138.69
	F12	0/1	0/1	3	-5	0	-	-	9	74.54	1/2	1/1	5	6	5	-	-	14	114.33	0/1	1/1	5	7	6	-	-	16	130.36
2	F4	0/1	1/1	5	5	0	-	-	13	106.46	0/1	1/1	5	7	4	-	-	15	122.39	0/1	1/1	5	7	5	-	-	16	130.25
5 goals	F6	0/1	0/1	3	-4	0	-	-	9	74.53	0/1	1/1	5	3	3	-	-	12	98.67	0/1	1/1	5	4	6	-	-	15	122.30
2 agents	F3	1/1	0/0	5	1	0	-	-	9	74.52	1/1	0/0	5	1	0	-	-	9	74.58	1/1	0/0	5	1	0	-	-	10	82.44
	F2	1/1	0/0	5	2	0	-	-	10	83.96	1/1	0/0	5	2	0	-	-	10	83.77	1/1	0/0	5	2	0	-	-	11	91.65
	F1	1/1	0/0	5	2	0	-	-	10	82.35	1/1	0/0	5	2	0	-	-	10	82.51	1/1	0/0	5	2	0	-	-	11	90.49
	F23	0/1	1/1	5	5	0	-	-	13	107.95	0/1	0/1	3	-2	0	-	-	6	64.02	0/1	1/1	5	4	5	-	-	14	116.00
	F1	1/1	0/0	5	1	0	-	-	9	74.79	1/1	0/0	5	1	0	-	-	9	74.45	1/1	0/0	5	1	0	-	-	10	82.95
	F11	0/1	0/1	3	-5	0	-	-	9	74.60	1/2	1/1	5	5	3	-	-	13	106.50	0/1	1/1	5	7	7	-	-	16	130.76
3	F6	0/1	0/1	3	-4	0	-	-	9	74.51	0/1	1/1	5	3	3	-	-	12	98.55	0/1	1/1	5	6	4	-	-	15	122.48
5 goals	F3	1/1	0/0	5	1	0	-	-	9	74.45	1/1	0/0	5	1	0	-	-	9	74.48	1/1	0/0	5	1	0	-	-	10	82.66
2 agents	F2	1/1	0/0	5	2	0	-	-	10	82.58	1/1	0/0	5	2	0	-	-	10	82.48	1/1	0/0	5	2	0	-	-	11	90.49
	F15	0/1	0/1	4	-1	0	-	-	9	74.39	0/1	1/1	5	4	3	-	-	12	98.43	0/1	1/1	5	3	4	-	-	13	106.48
	F10	1/1	0/0	6	1	0	-	-	10	83.80	1/1	0/0	6	1	0	-	-	10	84.21	1/1	0/0	6	1	0	-	-	10	83.94
	F1	1/1	0/0	6	1	0	-	-	10	82.62	1/1	0/0	6	1	0	-	-	10	82.63	1/1	0/0	6	1	0	-	-	10	82.76
4	F13	0/1	0/1	3	-5	0	-	-	9	74.69	1/2	1/1	6	5	5	-	-	14	114.68	0/1	1/1	6	6	3	-	-	15	122.67
6 goals	F11	0/1	0/1	4	-4	0	-	-	9	74.65	1/2	1/1	6	5	3	-	-	14	114.35	0/1	1/1	6	3	6	-	-	15	122.97
2 agents	F12	0/1	0/1	3	0	-7	-	-	9	76.18	0/1	0/1	3	0	-7	-	-	9	75.82	1/3	2/2	6	11	13	-	-	23	187.51
	F15	0/1	0/1	5	-1	0	-	-	9	74.67	0/1	1/1	6	4	3	-	-	13	106.74	0/1	1/1	6	2	4	-	-	13	107.16
	F6	0/1	0/1	5	0	0	-2	-	7	60.80	0/1	1/1	6	4	0	4	-	11	93.07	0/1	1/1	6	3	5	3	-	12	101.38
	F12	0/1	0/1	4	0	-3	0	-	7	59.70	2/3	1/1	6	5	5	0	-	12	102.64	0/1	1/1	6	6	4	4	-	13	109.20
5	F2	2/2	0/0	6	0	0	2	-	8	67.41	2/2	0/0	6	0	0	2	-	8	67.52	2/2	0/0	6	0	0	2	-	9	75.25
6 goals	F16	4/4	0/0	6	1	1	2	-	8	67.34	4/4	0/0	6	1	1	2	-	8	67.94	4/4	0/0	6	1	1	2	-	9	75.32
3 agents	F3	1/1	0/0	6	0	0	1	-	7	59.49	1/1	0/0	6	0	0	1	-	7	59.47	1/1	0/0	6	0	0	1	-	8	67.34
	F9	0/1	0/1	4	0	-3	0	-	7	59.31	1/2	1/1	6	4	3	0	-	11	91.43	0/1	1/1	6	5	6	4	-	13	107.38
	F20	0/1	0/1	6	0	-2	0	0	7	61.19	1/2	1/1	8	5	5	0	0	12	100.93	0/1	1/1	8	6	4	4	6	13	110.30
	F21	3/3	0/0	8	1	0	3	0	9	76.43	3/3	0/0	8	1	0	3	0	9	75.87	3/3	0/0	8	1	0	3	0	10	84.13
6	F18	2/2	0/0	8	0	0	3	0	9	76.46	2/2	0/0	8	0	0	3	0	9	76.12	2/2	0/0	8	0	0	3	0	10	84.47
8 goals	F9	0/1	0/1	6	0	-3	0	0	7	60.48	0/1	1/1	8	0	3	0	3	10	86.13	0/1	1/1	8	3	5	3	3	12	100.53
4 agents	F6	0/1	0/1	6	0	0	-4	0	7	59.87	0/1	1/1	8	3	0	2	0	10	84.44	0/1	1/1	8	3	2	5	3	12	100.27
	F4	1/1	0/0	8	0	2	0	0	8	68.37	1/1	0/0	8	0	2	0	0	8	68.40	1/1	0/0	8	0	2	0	0	9	75.96

configuration. The reactive and deliberative configurations reach all the goals except for the failures F23 in task 2 and F12 in task 4 of the reactive configuration. The reason is that MR is not able to create a multi-reactive solution because the agent is not capable to adapt its plan. Likewise as in the R1 configuration, the reactive repair achieves more goals in fewer cycles than the individual and deliberative configurations (92% of the failure executions).

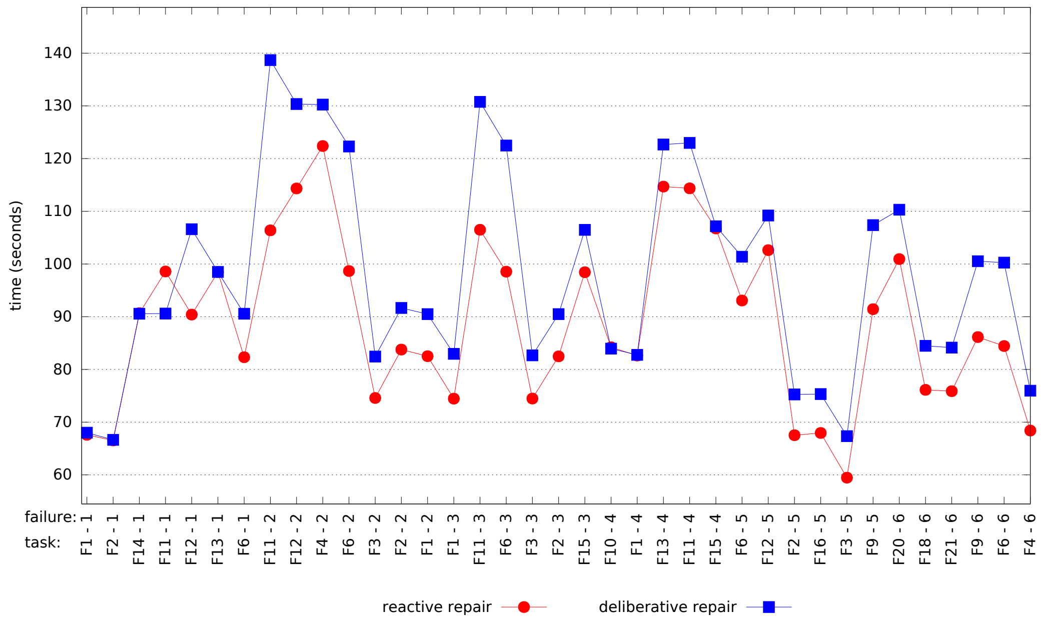

Figure 6.2: Real-time results when the reactive and deliberative repair configurations reach all the goals in the one-time scenario with R2 `rovers` configuration

Figure 6.2 compares the real-time results when the reactive and deliberative configurations reach all the goals in the one-time scenario with the R2 configuration. As in the R1 configuration, the reactive repair reaches all the goals in fewer cycles than the deliberative configuration except for the failure F11 of task 1, where the MR solution generates a collateral failure. In this failure, `rover2` loses its capability to seek for more soil samples, then `rover1` changes its position to seek for more soil samples that `rover2` needs to analyze. This solution generates a collateral failure when `rover1` finishes the execution of the multi-reactive solution and needs to execute its last action, because `rover1` is not in the required location to communicate the results of the rock analysis to the lander. The SR of `rover1` repairs the plan failure navigating to the required location.

Failures like F11 and F12 of task 2 provide us an idea of the good performance of our MARPE approach against the deliberative configuration. In the two failures, the MR solution generates a collateral failure due to `rover1` is not in the required location to communicate the results of the rock analysis to the lander when `rover1` finishes the execution of the multi-reactive solution. However, the reactive repair presents a better performance because its solution is shorter, in terms of actions, than the selected by the MD of the deliberative repair. For instance, in F12 of task 2 of the deliberative approach, `rover1` seeks for more soil samples (`rover2` loses its capability to seek for more soil samples) and `rover2` achieves its goals and the goals of `rover1`. Given that `rover2` performs its tasks as well as the ones of `rover1`, the plan becomes significantly longer.

Finally, in the cases where the differences are minimal, such as F1 and F2 of task 1, the failure is solved with our SR mechanism.

In summary, we concluded that adding more capabilities to the rovers, like in the R2 `rovers` configuration, also increases the reactive solutions, as for instance in F6 of task 1, where the failure is solved with the added rover-to-rover sample analysis communication capability. In addition, we also concluded that the repair and deliberative configurations present better results than the individual configuration.

6.4.1.2 two-times failure scenario

Table 6.7 depicts the results of applying two failures in a complete execution of the same six tasks of Table 6.4. As we highlight before, the *individual repair* configuration got the worst results. For this reason, we decide just to compare the repair configurations *reactive repair* and *deliberative repair*.

Table 6.7: Two-times failure scenario results with the R2 `rovers` configuration.

		reactive repair									deliberative repair								
		RP									RP								
task	fail	SR	MR	goals reached	delay				cycles	real time	SR	MD	goals reached	delay				cycles	real time
					(agents)					1:8 seg				(agents)					1:8 seg
					1	2	3	4						1	2	3	4		
	F1-F2	2/2	0/0	4	2	1	-	-	8	67.92	2/2	0/0	4	1	1	-	-	8	67.63
	F2-F6	1/2	1/1	4	5	3	-	-	11	91.91	1/2	1/1	4	4	2	-	-	11	91.66
1	F14-F7	2/3	1/1	4	3	6	-	-	13	106.49	2/3	1/1	4	3	5	-	-	12	100.68
4 goals	F11-F10	1/3	2/3	1	5	4	-	-	11	91.48	0/2	2/2	4	5	6	-	-	13	106.67
2 agents	F12-F4	0/1	2/2	4	10	7	-	-	16	131.58	1/3	2/2	4	12	5	-	-	19	154.51
	F13-F2	2/3	1/1	4	8	5	-	-	14	114.65	1/2	1/1	4	7	5	-	-	14	114.51
	F6-F3	0/1	2/2	4	6	5	-	-	12	98.57	1/2	1/1	4	4	5	-	-	12	98.43
	F11-F1	1/2	2/2	5	7	5	-	-	15	123.73	0/2	2/2	5	7	12	-	-	21	171.88
2	F12-F3	2/3	1/1	5	7	5	-	-	15	122.83	1/2	1/1	5	8	7	-	-	17	138.51
5 goals	F6-F3	0/1	2/2	5	6	5	-	-	14	115.69	1/3	2/2	5	0	7	-	-	18	146.25
2 agents	F3-F2	2/2	0/0	5	3	0	-	-	11	90.84	2/2	0/0	5	3	0	-	-	12	98.38
	F1-F11	2/3	1/1	5	4	3	-	-	12	99.79	1/2	1/1	5	8	2	-	-	17	140.01
3	F11-F1	1/2	1/1	5	5	3	-	-	13	106.59	0/2	2/2	5	8	10	-	-	19	154.90
5 goals	F3-F2	2/2	0/0	5	3	0	-	-	11	90.61	2/2	0/0	5	3	0	-	-	12	98.53
2 agents	F15-F3	0/1	2/2	5	7	5	-	-	15	122.58	1/3	2/2	5	5	1	-	-	16	130.41
	F10-F1	2/2	0/0	6	1	1	-	-	10	83.97	2/2	0/0	6	1	1	-	-	10	84.28
4	F1-F13	2/3	1/1	6	7	5	-	-	16	130.70	1/2	1/1	6	6	4	-	-	15	122.86
5 goals	F11-F1	1/2	2/2	6	7	5	-	-	16	130.61	1/3	2/2	6	7	5	-	-	18	146.43
2 agents	F15-F2	0/1	1/2	5	2	2	-	-	11	90.84	1/3	2/2	6	7	0	-	-	16	130.61
	F16-F3	4/4	0/0	6	1	1	3	-	9	77.02	4/4	0/0	6	1	1	3	-	10	85.01
5	F3-F15	1/2	1/1	6	0	0	5	-	11	93.22	1/2	1/1	6	0	0	5	-	11	93.64
6 goals	F12-F5	2/4	1/2	5	5	5	-1	-	12	103.62	1/2	1/1	6	6	6	8	-	15	123.44
3 agents	F9-F1	0/1	1/2	2	-1	-1	0	-	6	52.85	1/3	2/2	6	0	0	6	-	16	131.17
	F20-F10	1/2	1/1	8	6	5	0	0	13	110.20	1/3	2/2	8	3	6	1	2	16	133.95
6	F21-F22	5/5	0/0	8	1	1	3	1	9	77.10	5/5	0/0	8	1	1	3	1	10	86.54
8 goals	F1-F13	1/2	1/2	6	0	2	0	4	11	94.36	1/2	1/1	8	3	5	2	5	12	103.07
4 agents	F11-F1	0/1	1/2	6	0	-1	0	2	9	76.97	0/1	1/1	8	3	6	2	3	13	109.12

As we can see in Table 6.7, the two repair configurations are able to reach all the goals except in the reactive repair for failures F11-F10 of task 1, F15-F2 of task 4, F12-F5 and F9-F1 of task 5, and F1-F13 and F11-F1 of task 6. For instance, in F11-F10, the first failure F11 impacts over the `rover1` and it is repaired forming a multi-reactive solution with MR (the SR of `rover1` is not capable of repairing the plan failure because `rover1` lost the capability to seek for more rock or soil samples and the MR is activated), and when the agents finish the execution of the reactive solution the helper agent finds a collateral failure that repairs with the method SR. At the same time, the plan failure F10, which affects again the `rover1`, is executed and the agent `rover1` activates the SR of the reactive repair. SR is not capable of repairing the plan failure, and MR is activated. MR repairs the plan failure forming a new multi-reactive solution. However, during the execution of the reactive solution, another collateral failure appears and MR is activated again applying the recovery solution explained in Section 5.4.4.1. MR is not able to solve the collateral failure because it exceeds the maximum depth of the merge search space. Then, `rover1` and `rover2` end their plan execution. For F15-F2, the first failure F15 affect the `rover1`, which request for help to `rover2` and it is repaired forming a multi-reactive solution with the MR. Then, the plan failure F2 is executed and the MR works again applying the explained in Section 5.4.4.2. MR is not able to solve the failure during the execution of the reactive solution because `rover1` can not adapt its plan to the new failure solution. Then, `rover1` and `rover2` finish their plan execution. Same happens in F9-F1, but in this failure situations there is not solution plan because `rover1` loses the ability to analyze rock samples.

One failure situation where MR repairs the plan failure with the explained in Section 5.4.4.1 is in F6-F3 of task 2, where the first failure F6 affects `rover1`, i.e. `rover1` loses the ability to take image. Thus, the SR is activated, but SR is not capable of repairing the plan failure, and the agent activates the MR to request another agent to take the image. MR finds a solution plan where `rover2` takes the image and sends the results of the image to `rover1` which communicates the image to the lander. The second failure F3 appears during the execution of the multi-reactive solution. The failure F3 affects `rover2`, the helper agent, which solves it by activating the MR again.

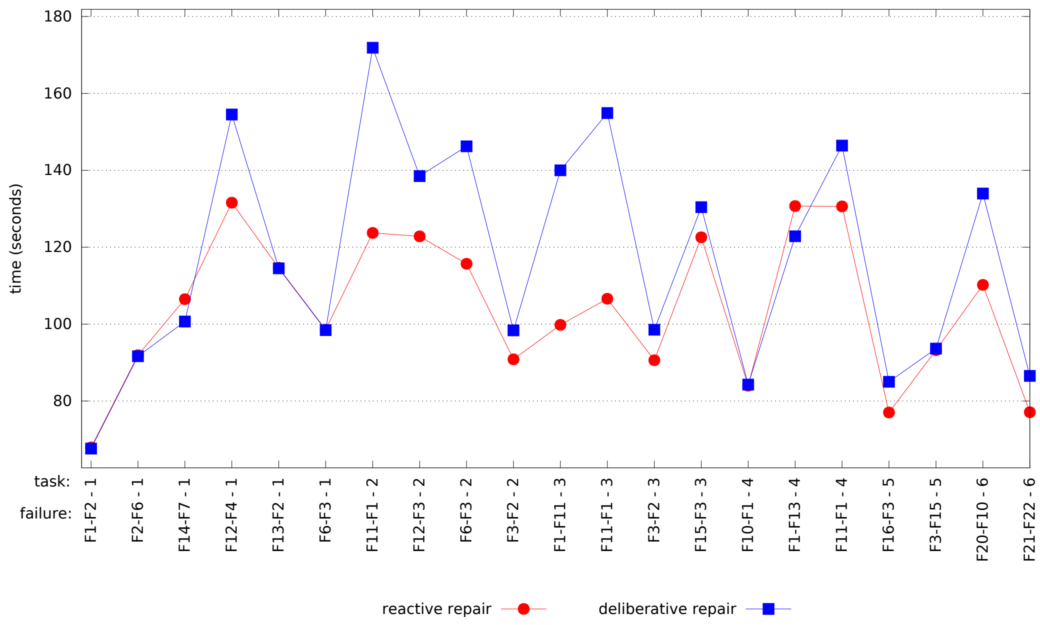

Figure 6.3: Real-time results when the reactive and deliberative repair configurations reach all the goals in the two-times scenario for the `rovers` domain

Figure 6.3 compares the real-time results when the reactive and deliberative configurations reach all the goals in the two-times scenario. The reactive repair also achieves all the goals in fewer cycles than the deliberative configuration excepts for the failures F14-F7 of task 1 and F1-F13 of task 4. In F14-F7 of task 1, the first failure F14 affects `rover1` and `rover2`, i.e. the camera of `rover1` loses calibration, and `rover2` loses the ability to seek for more samples. Thus, the SR of each rover is activated. The SR of each rover is able to find a solution plan and `rover1` and `rover2` continue with the execution of the new plan. Then, the second failure F7 is executed and the SR of `rover2` is activated. SR is not capable of repairing the plan failure, and MR of the reactive repair is activated to request another agent to seek for more soil samples. MR finds a solution plan where `rover1` seeks for more soil samples that `rover2` analyzes and communicates the results of the sample analysis to the lander. In contrast, with failure F7, the CP of the MD in the deliberative configuration decides that `rover1` executes part of the work of the `rover2`, i.e. `rover1` seeks for more soil samples, analyzes them and communicates the sample analysis to the lander. In F1-F13 of task 4, MR of the reactive configuration repairs the second failure with a solution plan where `rover2` seeks for more rock samples, analyzes them and communicates the sample analysis to `rover1`, which loses the sample result analysis together with the ability to seek for more rock samples; then, `rover1` communicates the results of the sample analysis to the lander. As we can see, in this solution, `rover2` performs part of the work of `rover1`, in contrast with the solution of MD in the deliberative configuration where `rover2` executes the complete work of `rover1`. Finally, in some cases where the differences are minimal, such as F1-F2 of task 1, or F10-F1 of task 4, the failure is solved with the SR.

A higher degree of equilibrium in the agents delay can be defined as the situation where all the agents participate actively in the solution plan, reaches the goal more cooperatively, and additionally have a minimal delay. Our approach reaches a higher equilibrium in the agents delay, in opposition to the MD approach, where sometimes one agent do all the work, and the another agent finishes earlier, such as the rover1 in the failure F15-F3 of task 3. In this failure situation, the MD generates a solution plan in which the rover2 performs all the work, and the rover1 has only to wait. Same happens in F20-F10 of task 6.

6.4.2 `elevators` domain

The multi-agent `elevators` domain differs from the single-agent domain (see Appendixes C.2.1 and C.1, respectively) that we select the `elevator` as type `agent`. We used the STRIPS version of the IPC without actions costs and temporal. Our `elevators` domain presents some differences from the IPC version.

- We remove the predicates related to the maximum number of passengers allowed in an elevator, i.e. the predicate (`can-hold` ?lift - `elevator` ?n - `count`) and the associated predicates (`passengers` ?lift - `elevator` ?n - `count`) and (`next` ?n1 - `count` ?n2 - `count`), which are used to know the current number of passengers and to increase/decrease a passenger in an elevator, respectively. The reason is that simulating a failure that changes the position of a passenger to be inside or outside the elevator would imply to previously check the maximum and minimum capacity conditions of the elevator in order to ensure the failure is a consistent change in the domain.

- We add the function (`working-lift ?l`) - `option`, and the predicate (`door-working ?l ?f`) to specify whether the elevator `?l` is working, and if the door of the elevator `?l` is operative or not in the floor `?f`, respectively. These two fluents allow us to promote a higher degree of cooperation between agents because, for instance, if the door of the elevator is not operative then the elevator needs to request for help in order to transport the passenger to the target floor.

In this domain, an `elevator` agent collaborates in the multi-repair solution employing a total number of two capabilities: board a passenger in a floor, and leave a passenger in a floor.

Table 6.8: Failures generated for the `elevators` domain

	failure	description
single failures	F1	the `passenger` is on another `floor`.
	F2	the `passenger` is in an specify `elevator`.
	F3	the `elevator` changes its position to another `floor`.
	F4	the `elevator` is broken due to a hardware failure.
	F5	the elevator's door is damaged in an specific `floor`.
	F6	the `elevator` is not able to reach the `floor`.
	F7	F1 and F5.

Table 6.8 shows all the failures we generated for the `elevators` domain, which

we divided into single failures (F1 until F6) and the compound failure, F7. Like the `rovers` domain, a single failure alters a fluent in the current world state like failure F3, in which the `elevator` changes its position to another floor. The compound failure F7 changes two or more fluents in the current world state. We randomly chose and applied a single or compound failure from Table 6.8 during the simulated plan execution. The elevators' actions for solving the failures can be any combination of this set of actions:

1. the `elevator` moves up to one different `floor`
2. the `elevator` moves down to one different `floor`
3. the `elevator` boards the `passenger` in the specific `floor`
4. the `elevator` leaves the `passenger` in the specific `floor`

For instance, assuming we apply the single failure F1, the solution plan of the elevator is to moves up to the specific floor, boards the passenger, moves up or down to the target floor, and leaves the passenger. Notice that some failures, such as F4 and F5, do not have any self-repair solution; i.e. they require a multi-repair solution plan.

Table 6.9 shows the results of the `elevators` domain with the one-time failure scenario. As we can see, the six tasks that we executed of the `elevators` domain considers the following planning tasks:

- the task 1 has two `elevators`, and three goals
- the task 2 has two `elevators`, and three goals
- the task 3 has two `elevators`, and four goals
- the task 4 has two `elevators`, and four goals
- the task 5 has three `elevators`, and five goals

Table 6.9: Results for the `elevators` domain applying one-time failure.

		reactive repair									deliberative repair								
		RP									RP								
task	fail	SR	MR	goals reached	delay				cycles	real time	SR	MD	goals reached	delay				cycles	real time
					(agents)					1:8 seg				(agents)					1:8 seg
					1	2	3	4						1	2	3	4		
	F1	1/1	0/0	3	0	2	-	-	9	75.53	1/1	0/0	3	0	2	-	-	11	91.58
	F2	1/2	1/1	3	3	5	-	-	12	98.57	0/1	1/1	3	4	2	-	-	13	106.75
1	F4	0/1	0/1	2	-2	0	-	-	7	58.65	0/1	1/1	3	4	3	-	-	13	106.68
3 goals	F5	0/1	0/1	2	-3	0	-	-	6	50.50	0/1	1/1	3	3	0	-	-	12	98.46
2 agents	F7	0/1	1/1	3	3	0	-	-	12	98.56	0/1	1/1	3	5	0	-	-	14	114.48
	F1	1/1	0/0	3	2	0	-	-	10	83.52	1/1	0/0	3	2	0	-	-	10	83.54
2	F2	0/1	1/1	3	5	2	-	-	13	106.63	0/1	1/1	3	1	3	-	-	11	91.57
3 goals	F5	0/1	0/1	2	-3	0	-	-	5	42.75	0/1	1/1	3	2	3	-	-	11	90.65
2 agents	F7	1/2	1/1	3	5	5	-	-	13	106.68	0/1	1/1	3	0	5	-	-	13	106.56
	F3	1/1	0/0	4	1	0	-	-	10	83.62	1/1	0/0	4	1	0	-	-	10	82.50
3	F2	0/1	1/1	4	3	2	-	-	12	98.68	0/1	1/1	4	1	3	-	-	12	98.43
4 goals	F6	0/1	0/1	3	-4	0	-	-	9	75.76	0/1	1/1	4	5	3	-	-	14	115.80
2 agents	F7	0/1	1/1	4	4	4	-	-	13	106.38	0/1	1/1	4	6	5	-	-	15	122.42
	F3	1/1	0/0	4	1	0	-	-	10	83.53	1/1	0/0	4	1	0	-	-	10	82.31
4	F2	1/1	0/0	4	0	0	-	-	8	66.66	1/1	0/0	4	0	0	-	-	9	74.43
4 goals	F5	0/1	0/1	2	0	-3	-	-	10	81.60	0/1	1/1	4	3	5	-	-	14	114.41
2 agents	F7	0/1	1/1	4	4	5	-	-	13	106.57	0/1	1/1	4	5	6	-	-	15	122.35
	F1	1/1	0/0	5	0	0	1	-	9	74.43	1/1	0/0	5	0	0	1	-	9	74.94
5	F2	0/1	1/2	2	-2	0	-3	-	6	51.15	0/1	1/1	5	3	0	1	-	11	90.91
5 goals	F5	0/1	0/1	4	0	-1	0	-	8	69.69	0/1	1/1	5	3	5	2	-	13	107.15
3 agents	F7	0/1	2/2	4	5	-5	0	-	13	107.50	0/1	1/1	5	5	6	3	-	14	115.15
	F1	2/2	0/0	5	0	0	2	-	10	85.75	2/2	0/0	5	0	0	2	-	10	83.46
6	F2	1/1	0/0	5	-1	0	0	-	8	67.16	1/1	0/0	5	-1	0	0	-	8	66.74
5 goals	F5	0/1	0/1	4	0	-1	0	-	8	69.24	0/1	1/1	5	3	5	2	-	13	107.20
3 agents	F7	1/2	1/1	5	3	5	0	-	11	91.45	0/1	1/1	5	4	5	2	-	13	106.99

- the task 6 has three `elevators`, and five goals

Like the `rovers` domain, the deliberative repair configuration is the only one capable of achieving the goals of all the tasks because the CP of the MD repairs the failures with a global vision of the world state. For instance, in F4 of task 1, `elevator1` presents a hardware failure and activates the SR of the reactive repair. SR is not capable of repairing the plan failure, and MR is activated to request another agent to fix the hardware plan failure, which is not possible because there is not action that fix hardware failures. In contrast, with the deliberative repair, the CP of the MD generates a new plan where the `elevator2` executes all the work of `elevator1`.

On the contrary, the reactive repair is not able to reach all the goals in the 32% of the execution failures. It occurs for two reasons: 1) the current plan window is too small or 2) it requires joint actions to fix the failure, and consequently more interactions between the agents. For instance, failure F5 of task 1 represents case 1). In this failure, `passenger1` can not board in the `elevator1` because the door of `elevator1` has a hardware failure in `floor8`. Thereby, it is not possible to execute the next action "5.0: `(board elevator1 passenger1 floor8)`" (see Appendix C.2.2.1). `elevator1` is covering a plan window of two actions, i.e. the actions "5.0: `(board elevator1 passenger1 floor8)`" and "6.0: `(move-down elevator1 floor8 floor4)`", any activation of MR will request to fix the hardware failure of the door, which is not affordable. In contrast, If the plan window is higher than two, it will cover the three final actions, and then it will be possible to find a solution plan where another `elevator` transports `passenger1` to the destination floor. The same failure F5 but in task 5 represents the case 2), `elevator2` detects a hardware failure and activates the SR. SR is not capable of solving the plan failure, and MR is activated to request another `elevator` agent to transport the `passenger2` to the target `floor8`. The solution requires joint actions that employs a higher number of interactions between agents because `elevator2` needs to move to another floor in which elevator's door is working, and leaves the passenger2; then, another elevator should board and transport the passenger2 to the target floor8. In contrast, the CP of the MD generates the required solution plan.

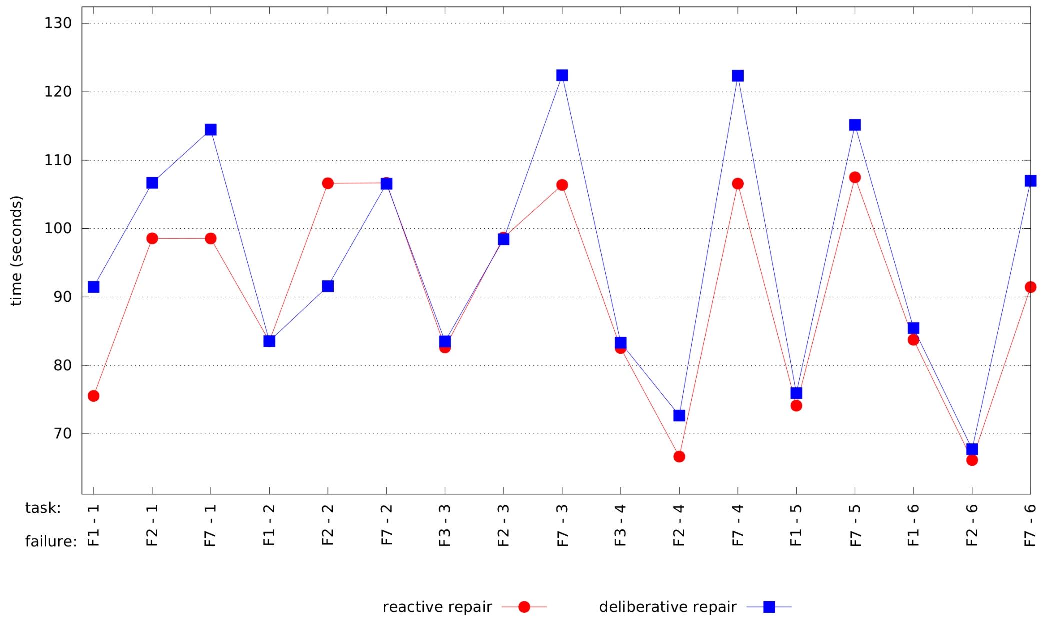

Figure 6.4: Real-time results when the two repair configurations reach all the goals for the elevators domain

Figure 6.4 shows the real-time results when the two repair configurations reach all the goals. Clearly, the reactive repair achieves all the goals in fewer cycles than the deliberative configuration excepts for F2 of task 2. In this failure, elevator1 detects that the passenger1 took the wrong elevator2 in the floor0 and activates the SR. SR is not capable of repairing the plan failure and activates the MR to request the elevator2 to leave the passenger1 in the floor0; unlike, the MD of the deliberative repair that generates a solution plan where the elevator2 reach the complete goals of elevator1.

In cases where the differences are minimal, such as F1 of task 2 or F3 of task 4, the failure is solved with our SR mechanism. Consequently, as it is shown in

Table 6.9, the deliberative repair also presents fewer delay in the agents' plan execution, such as the failure F7 of task 3 where the agents in the reactive repair present a less number of delay in the execution cycles than the deliberative.

6.4.3 `transport` domain

The multi-agent `transport` domain presents some differences in relation to the single-agent version (see Appendixes D.2.1 and D.1, respectively).

1. We add the predicates (`engine-operating ?t`) and (`hoist-operating ?t`) to specify whether the motor of the truck `?t` and the crane of the truck `?t` are working or not, respectively. Thus, if the motor of `?t` is not operative, `?t` will not be able to drive to any location and it will request help to other trucks. In the same way, if the crane of `?t` is not operative, `?t` will not be capable to load or `unload` any package and the only possible solution will be to resort for help to other trucks. In other words, any failure in the two new predicates will affect directly the actions `drive`, `load`, and `unload` with not possible self repairing solution, forcing the failing agent to activate the MR method in the reactive repair configuration or the MD method in the deliberative repair configuration.

2. We also endow `truck` agents with the abilities to load a `package` inside another `truck` and unload a `package` from another `truck`. This new abilities are included to promote an alternative joint way of repairing failures when the crane of the truck is not operative. For instance, if the crane of one `truck` is not operative to unload a `package` to an specific location, the `truck` activates the MR method. With the new abilities, another `truck` may drive to the specific location in order to unload the `package` from the failing `truck`, and transport it to the specific location.

Table 6.10: Failures generated for the `transport` domain

	failure	description
single failures	F1	`truck` changes its position to another `location`.
	F2	`truck` loses the good maps to drive from one `location` to another `location`.
	F3	`truck` engine is not operative.
	F4	`truck` hoist is not operative.
	F5	`package` changes its position to one `location` or `truck`.

Table 6.10 shows all the possible failures we generated for the `transport` domain. All the failures are single failures that alters a fluent in the current world state like F2, in which the `truck` loses the good maps to travel from one location to another location. In each execution of this domain, we simulated a compound failure by randomly choosing and applying two single failures from the Table 6.10 during the plan execution. The first single failure is F3 or F4 because we want to ensure the activation of the methods MR and MD. The second failure will be any single failure of the Table 6.10. Thus, our compound failures are any combination of the form F3-Fi or F4-Fi, where Fi is one of the single failures F1, F2, F3, F4 or F5. Thus, in this domain, `trucks`' actions for collaborating in the multi-repair solution can be any combination of this set of actions:

1. the `truck` drives to the specific `place`
2. the `truck` loads the `package` from the specific `location`
3. the `truck` loads the `package` from another specific `truck`
4. the `truck` unloads the `package` at the specific `location`
5. the `truck` unloads the `package` to another specific `truck`

For instance, assuming we apply the failure F1, the solution plan of the `truck` is to drive to the particular position, load the `package`, drive to the target position and unload the `package`. If we apply the failure F4, the `truck` will not be capable of self-fixing the plan failure, and activates the MR to request help to other `truck` that can drive to the specific location, and load/unload the package from/to the failing truck.

Table 6.11 shows the results for the `transport` domain with the one-time failure scenario. As we can see, the six tasks that we executed are the follows:

- our task 1 has two `trucks` and two goals
- our task 2 and 3 has two `trucks`, and three goals
- our task 4 has two `trucks`, and four goals
- our task 5 and 6 has three `trucks`, and five goals

The deliberative repair configuration is the only one capable of achieving the goals of all the tasks except in the execution failure F3-F4 of task 2. The reason is that the system randomly selects the two failures F4 and F3 to form a compound failure that affects `truck1` and `truck2`, respectively. Hence, `truck1` detects the crane is not working to load and unload packages and activates the SR. SR is not capable of repairing the plan failure and activates the MD of the deliberative configuration. Then, the `truck2` interrupts its plan execution, and the two agents communicate their current state to the CP of the MD, but the MD is not capable of fixing the plan failure due to `truck2` presents also a failure that can not be repaired, the motor of `truck2` is not working. The same reason applies for the MR of the reactive repair configuration.

Another remarkable difference is that the SR in the `transport` domain repairs fewer failures than the `rovers` and `elevators` domains, and thereby, the MR and MD are activated more often in this domain. As we explain before, this happens because the system randomly selects the single failures F3 or F4, which have not possible single repairing solution. For instance, in failure F4-F2 of task 1, the crane of `truck1` is not operative (F4), and `truck2` loses the good maps to drive from location `s2` to `s0` (F2). `truck1` detects the plan failure and activates the SR in the

Table 6.11: Results for the `transport` domain with the one-time failure scenario.

		reactive repair									deliberative repair								
		RP									RP								
task	fail	SR	MR	goals reached	delay				cycles	real time	SR	MD	goals reached	delay				cycles	real time
					(agents)					1:8 seg				(agents)					1:8 seg
					1	2	3	4						1	2	3	4		
	F3-F2	0/1	1/1	2	4	3	-	-	8	67.35	0/1	1/1	2	1	5	-	-	9	75.46
	F3-F1	0/1	1/1	2	4	3	-	-	8	66.44	0/1	1/1	2	1	5	-	-	9	74.40
1	F4-F1	0/1	1/1	2	4	1	-	-	8	66.35	0/1	1/1	2	4	2	-	-	8	66.46
2 goals	F4-F2	0/1	1/1	2	2	2	-	-	6	50.50	0/1	1/1	2	3	3	-	-	7	58.61
2 agents	F3-F2	0/1	1/1	2	4	4	-	-	8	66.28	0/1	1/1	2	5	5	-	-	9	74.31
	F4-F1	0/1	1/1	2	4	4	-	-	8	66.56	0/1	1/1	2	4	4	-	-	8	66.42
	F3-F1	1/2	1/1	3	4	4	-	-	12	99.39	1/2	1/1	3	4	4	-	-	12	99.51
	F4-F2	0/1	1/2	0	-2	2	-	-	6	50.47	0/1	1/1	3	3	1	-	-	11	90.74
2	F4-F5	0/1	1/1	3	4	5	-	-	12	98.35	0/1	1/1	3	2	4	-	-	12	98.56
3 goals	F3-F2	0/1	1/1	3	3	4	-	-	11	90.43	0/1	1/1	3	3	4	-	-	11	90.45
2 agents	F4-F1	0/1	0/1	1	-7	0	-	-	4	34.36	0/1	1/1	3	2	1	-	-	10	82.43
	F3-F4	0/1	0/1	1	-4	0	-	-	4	34.82	0/1	0/1	1	-3	-3	-	-	5	42.71
	F3-F2	0/1	1/2	0	-4	-2	-	-	3	27.94	1/2	1/1	3	5	4	-	-	12	99.51
	F4-F1	0/1	1/1	3	4	3	-	-	11	90.60	0/1	1/1	3	5	4	-	-	12	98.51
3	F4-F3	0/1	1/1	3	4	3	-	-	11	90.42	0/1	1/1	3	4	4	-	-	11	90.54
3 goals	F4-F2	0/2	2/2	3	9	5	-	-	16	130.41	0/1	1/1	3	5	3	-	-	12	98.53
2 agents	F4-F2	0/2	2/2	3	7	3	-	-	14	114.46	0/1	1/1	3	5	5	-	-	12	98.66
	F3-F2	0/1	0/1	2	0	-6	-	-	5	43.42	0/1	1/1	4	2	3	-	-	11	91.71
	F3-F2	0/1	1/1	4	3	4	-	-	12	98.70	0/1	1/1	4	3	5	-	-	12	98.93
4	F4-F3	0/1	0/1	2	0	-5	-	-	5	42.47	0/1	1/1	4	2	4	-	-	12	98.50
4 goals	F4-F1	0/1	1/1	4	1	3	-	-	11	90.47	0/1	1/1	4	3	2	-	-	11	90.54
2 agents	F3-F2	0/1	1/1	4	4	3	-	-	11	90.54	0/1	1/1	4	4	5	-	-	12	98.26
	F4-F1	1/3	2/2	5	6	5	1	-	13	108.41	1/2	1/1	5	7	6	3	-	14	116.06
	F3-F2	0/1	1/1	5	0	3	3	-	9	75.23	0/1	1/1	5	0	4	4	-	10	83.19
5	F4-F2	0/2	1/2	4	3	5	0	-	10	83.44	0/1	1/1	5	4	6	3	-	12	99.04
5 goals	F3-F4	0/2	1/2	3	-2	4	4	-	9	74.93	0/1	1/1	5	7	5	4	-	14	115.23
3 agents	F4-F5	0/1	1/1	5	0	2	2	-	7	59.17	0/1	1/1	5	0	1	1	-	8	66.94
	F4-F1	1/2	1/1	5	3	3	1	-	10	84.19	1/2	1/1	5	4	4	5	-	14	116.08
	F3-F2	0/1	1/1	5	4	4	0	-	9	75.28	0/1	1/1	5	6	6	0	-	13	107.34
6	F4-F5	1/2	1/1	5	4	3	2	-	11	91.28	1/2	1/1	5	2	1	0	-	11	92.26
5 goals	F3-F2	1/2	0/1	2	1	0	-5	-	6	50.89	1/2	1/1	5	6	0	7	-	16	131.17
3 agents	F4-F5	0/1	1/2	3	-1	1	0	-	9	75.28	0/1	1/1	5	4	2	0	-	13	107.10

two repair configurations. SR is not capable of repairing the plan failure, and the MR and MD are activated in the reactive and deliberative repair, respectively. MR repairs the plan failure by requesting another agent to unload `package1` and to leave it in the location `s1`. The MD also repairs the plan failure with the CP.

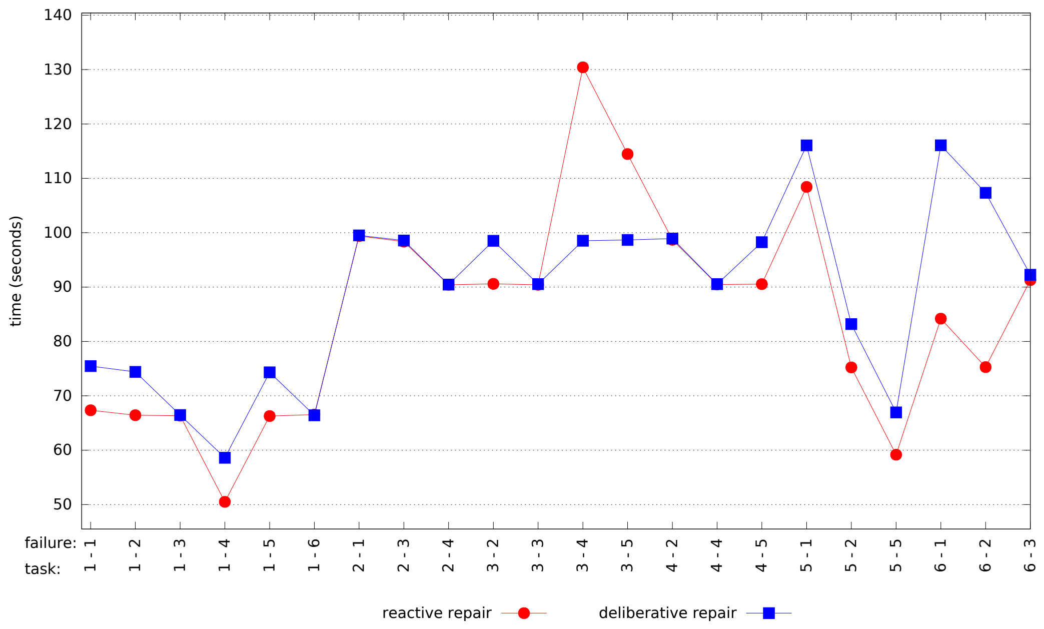

Figure 6.5: Real-time results when the reactive and deliberative repair configurations reach all the goals for the `transport` domain

Figure 6.5 compares the real-time results when the reactive and deliberative configurations reach all the goals, i.e. in the 69% of the total number of execution failures. The reactive repair is not able to reach all the goals in 31% of the total number of execution failures. However, the reactive repair presents a better performance achieving the goals in fewer cycles than the deliberative configuration except in two failures F4-F2 of task 3, where the MR solution generates a collateral failure. In the first failure F4-F2 of task 3, the crane of `truck1` is not operative and `truck2` loses the good map to drive from location `s0` to `s2`. The MR of the reactive repair generates a plan solution in which `truck2` using its crane loads the `package1` inside the `truck1`. This plan solution generates a collateral failure when `truck1` the execution of the collaborative solution finishes and needs to execute the remaining actions of the original plan to unload the `package1` (the crane of `truck1` previously presented a hardware failure). Then, as the trucks are not collaborating together, the SR is activated. SR is not capable of fixing the plan failure and activates the MR. MR generates a plan solution where `truck2` drives to the specified location, loads the packages from the `truck1`, and unloads them to the specified location `s1`. In contrast, the MD generates a plan solution where the `truck2` executes all the work of `truck1`. The same happens in the second failure F4-F2 of task 3.

To sum up, failures F4-F1 and F3-F2 of task 6 gives us an idea of the good performance of our approach against the deliberative configuration. For instance, in failure F4-F1 of task 6, the crane of the `truck1` presents a hardware failure and `truck3` changes its position from `s1` to `s2`. First, `truck3` detects the plan failure F1 and activates the SR in the two repair configurations. SR fix the failure by driving `truck3` from `s2` to `s1`. Then, in the next execution cycle, `truck1` detects the plan failure F4 and activates the SR in the two repair configurations. SR is not capable of repairing the failure and activates the MR in the reactive repair configuration and the MD in the deliberative configuration. MR repairs the plan failure with the best and least conflictive solution, `truck2` drives to the specific location `s1`, load the package and transport it to the target location `s2`. In contrast, the CP of the MD generates a solution plan where `truck3` transport the package to the specific location `s1`. In terms of metric, this solution is worst than the selected by the MR. The reason is that the `truck2` is closer to the specific location than `truck3`, and one of its goals is to transport another package to the same target location. Then, it is more reasonable to fix the plan failure with the `truck2`.

In cases where the differences are minimal in Table 6.11, the deliberative repair also presents fewer delay in the agents' plan execution, such as the failures F4-F1 and F4-F3 of task 3 where the agents in the reactive repair present a less number of delay in the execution cycles than the deliberative configuration.

6.5 Conclusions

In this chapter, we have presented the evaluation of our MARPE model. First, we presented the repair configurations we used in the experiments. Next, we introduced the domains and the specification of the multi-agent planning tasks. Finally, we analyzed the obtained results by comparing principally our MARPE model, which we used in the reactive repair configuration, against a centralized deliberative method, which we employed in the deliberative repair configuration. We used the single repair planning mechanism as a basis in all the configurations.

The results in Section 6.4 show that the MARPE model presents an outstanding performance by achieving more goals in fewer cycles than the deliberative repair in a complete execution simulation of the planning tasks. The most relevant limitation of our model is that some recovery solutions can generate collateral failures in the plans of the helper agent. However, the excellent performance of our MARPE model minimizes this limitation because whenever a collateral failure is detected the single repair or the Collaborative Repair handle to repair it.

The exhaustive experimentation carried out on several non-coordinated planning domains with the collaborative repair mechanism confirms that the MARPE model with its Collaborative Repair process is a very suitable multi-agent mechanism to fix failures that represent slight deviations from the main course of plan actions. The results support several conclusions: the accuracy of the repairing structures to recover failures, the reliability and performance of our multi recovery search procedure in comparison with either a central deliberative mechanism or a single reactive mechanism, and the focus that it is more beneficial to repair small plan windows rather than the whole plan.

Chapter 7

Conclusions and future works

"The afternoon knows what the morning never suspected."

(Robert Frost)

In this chapter, we present the general conclusions of this Ph.D. dissertation and some future work. The chapter is organized into two sections. The first section describes our Collaborative Repair approximation and enumerates the more relevant contributions. The second and last section points at several future research lines directly related to this work.

7.1 Conclusions

This work has focused on the development of a multi-agent reactive planning and execution model that would endow robot agents with plan monitoring and executing facilities as well as the machinery necessary to allow a rover to recover from a plan failure by itself or with the collaboration of other execution agents. This type of requirements are not easily satisfied by current reactive planners and multi-agent architectures.

7.1.1 Proposed approximation

The proposed solution starts from a planning and execution architecture, where we associated each execution agent with a planning agent. Although we designed the architecture to include planning and execution agents, during the development of this Ph.D. dissertation we only focused on the execution agents. In our architecture, execution agents are capable of executing and repairing plans within the same environment. Each execution agent has its planning task with its goals and calls its associated planning agent which generates its initial plan. Plans are independent of each other, but they all need to be successfully executed to solve the overall planning task; i.e., the goals of all the execution agents need to be fulfilled. The principal objective of the architecture is to define a framework in which the execution process is not only an isolated executor but a process capable of applying repairing techniques. For the execution agents, we developed a single-reactive planning technique to take corrective actions quickly and a multi-agent reactive planning technique to avoid demanding a new plan from the planning agent.

The main contribution of this work is, undoubtedly, the developed Reactive Planner, which we integrated into the execution agents. The Reactive Planner is designed along two fundamental stages: one for a single-agent plan failure recovery and another for multi-agent plan failure recovery. In the following, we highlight the most relevant aspects of each individual contribution.

7.1.1.1 Repairing structures

We proposed to use repairing structures as search trees, which allow execution agents to recover from plan failures. Our repairing structures work on a portion of the plan. We calculate them at runtime with a time-bounded limited process that exploits Machine Learning techniques to estimate the size of the search tree that the agent can generate without exceeding a fixed time limit.

7.1.1.2 Self-repairing process

We put forward a new general reactive repair technique that quickly resumes a portion of an agent's plan. The self-repair technique is built upon the repairing structures, and it simply requires performing a modified breadth-first search on the tree. Our repairing structure always guarantees the optimal solution.

7.1.1.3 Multi-agent repairing process

As far as we are concerned, there is no reactive repair system in which several execution agents participate together to solve plan failures. The work on this Ph.D. dissertation is the first one that proposes a multi-agent repair process in which an agent requests help from other agents to repair a plan failure. Our multi-agent model is not a solution to cooperatively solve all the upcoming failures but a system towards a global reactive solution. Thus, to improve responsiveness, only a maximum of two agents intervene in the collaborative repair solution, namely the agent that fails and the agent that provides the help. The more agents involved, the more time-consuming the generation of the recovery solution. Even though this clearly restricts the failures that can be reactively solved, for failures that require several agents involved in the repair, it is usually more worthy resorting to a replanning solution.

In our repair model, we proposed to recover from a plan failure by calculating a new time-bounded repair structure, which includes the goals of the plan window of the helper agent and the recovery state of the failing agent. The failing agent receives the proposed recovery solution from the helper agent and combines it into a newly integrated search-tree. If this is viable, then the agents will be able to undertake a collaborative reactive solution within a limited time.

7.1.2 Contributions

We conducted the objective of this Ph.D. dissertation towards the proposal and evaluation of a reactive multi-agent technique, suitable for the resolution of certain types of real problems, characterized by the dynamism of the environment, their limited reaction times, and the limitations when communicating with the deliberative agent. Despite the complexity of working under these requirements, we satisfactorily developed a collaborative execution and planning model to solve various planning tasks with these characteristics. From our point of view, we favorably met the objectives of this Ph.D. dissertation. Below, we listed the most general contributions of this work:

1. Design and development of a single-agent planning and execution architecture that comprises a general planning and execution model that endows an execution agent with plan monitoring, execution, deliberative planning, and reactive planning capabilities.

2. Development of a domain-independent self-reactive planner capable to perform a time-bounded process. We exploit Machine Learning techniques to promptly create repairing structures that operate on the plan to fix problems at execution time.

3. Evaluation of the developed self-reactive planner validating the generation of the repairing structures within the available time, the best-suited regression model, the recovery of failures due to slight deviations of the main course of the plan actions, and the usefulness of the single repairing structure to fix more than one action in a plan fragment.

4. Extension of the single-agent architecture to a multi-agent planning and execution framework, which includes several features, such as, i) supporting to execute several agents in the same environment, ii) providing agents with communications capabilities to allow information exchange between them, iii)

simulation of the state of the world, and iv) the capacity to add new modules easily for the agents.

5. Implementation of the multi-agent repair mechanism inside the reactive planner with the principal objective of ensuring the continuous and uninterruptedly flow of the execution agents. The main characteristic of this repair mechanism is the repairing structures and their integration with the actions of two different agents.

6. Evaluation of the developed multi-agent Reactive Planner checking its stable performance and some limitations that we found.

The work carried out in this Ph.D. dissertation initiated with a short stay (**EEBB-112-04550**) executed at NASA Ames Research Center during a period of six months with the Dr. Jeremy Frank as my advisor and mentor. The aim was to endow Mars planetary rovers not only with a self-repair process, but with a collaborative process to allow rovers or spacecraft to minimize time communication to Earth, which normally has a long delay. Firstly, we incorporated the PELEA architecture (explained in the Section 2.1.1.6) inside the Mars rovers (execution agents), so that, each rover autonomously generates and executes its plan. With the limitation that they can not communicate between them. Secondly, we developed a simplified and reactive version of a planning and execution architecture for multiple agents, PlanInteraction, designed for simulated or real executions (explained in detail in Section 2.1.2.3). Specificaly, we focus on providing the rovers with capabilities of self-executing, monitoring and repairing, and also with communication between them. Next, we implemented a conflict resolution mechanism for the Mars domain. The conflict resolution mechanism had two main intentions:

- it allowed to coordinate the actions of two rovers to fix potential conflicts that may arise during the execution of a plan. For this, we employed a bilateral protocol of negotiation of multiple variables. The protocol allowed to negotiate which agent will first use a shared variable to execute its action.

- If there is no agreement, the rovers continue with their plan execution as normal and repair the failures by calling its own planner or the central planner on Earth whenever it occurs.

Finally, we researched in the field of plan monitoring. We developed a framework based on templates that monitors the changes in the variables during the execution of the plan. We studied the problem of extracting, from the domain and the plan, the variables to observe during the plan execution. The process works in two phases: First, we automatically compile to Metric Temporal Logic (MTL) an encoding the restrictions to be monitored. Second, we use known algorithms from the runtime verification community to efficiently compile monitoring processes from MTL. The monitoring processes are efficient in the sense that they allow to include time restrictions and quickly detect deviations from the plan.

The results of the short stay and the whole work of this thesis have led to a series of publications, which we referenced throughout the memory. Of these, the following stand out:

- César Guzmán, Pablo Castejon, Eva Onaindia, and Jeremy Frank. Reactive execution for solving plan failures in planning control applications. *Journal of Integrated Computer-Aided Engineering*, 22(4):343–360, 2015.

- César Guzmán, Pablo Castejon, Eva Onaindia, and Jeremy Frank. Robust plan execution in multi-agent environments. In 26th *IEEE International Conference on Tools with Artificial Intelligence* (ICTAI), pages 384–391, 2014.

- César Guzmán-Alvarez, Pablo Castejon, Eva Onaindia, and Jeremy Frank. Multi-agent reactive planning for solving plan failures. In *Hybrid Artificial Intelligent Systems* - 8th International Conference, HAIS 2013. Volume 8073 of Lecture Notes in Computer Science, pages 530–539. Springer, 2013.

- Thomas Reinbacher, César Guzmán. Template-Based Synthesis of Plan Execution Monitors. In *Hybrid Artificial Intelligent Systems* - 8th International Conference, HAIS 2013. Volume 8073 of Lecture Notes in Computer Science, pages 451–461. Springer, 2013.

- César Guzmán, Vidal Alcazar, David Prior, Eva Onaindia, Daniel Borrajo, Juan Fdez-Olivares, and Ezequiel Quintero. Pelea: a domain-independent architecture for planning, execution and learning. In *ICAPS 6th Scheduling and Planning Applications woRKshop* (SPARK), pages 38–45, 2012.

- César Guzmán-Alvarez, Vidal Alcazar, David Prior, Eva Onaindia, Daniel Borrajo, Juan Fdez-Olivares. Building a Domain-Independent Architecture for Planning, Learning and Execution (PELEA). 21th *International Conference on Automated Planning and Scheduling* (ICAPS) - Systems Demo. pages 27-30, Freiburg (Germany), 2011

- Ezequiel Quintero, Vidal Alcazar, Daniel Borrajo, Juan Fdez-Olivares, Fernando Fernandez, Angel Garcia-Olaya, César Guzmán-Alvarez, Eva Onaindia, David Prior. Autonomous Mobile Robot Control and Learning with PELEA Architecture. AAAI-11 *Workshop on Automated Action Planning for Autonomous Mobile Robots* (PAMR). pages 51-56, San francisco (USA), 2011.

- Antonio Garrido, César Guzmán, and Eva Onaindia. Anytime plan-adaptation for continuous planning. In 28th *Workshop of the UK Planning and Scheduling Special Interest Group* (PlanSIG'10), Brescia (Italia), 2010.

- PELEA: Planning, Learning and Execution Architecture. Vidal Alcazar, César Guzmán-Alvarez, David Prior, Daniel Borrajo, Luis Castillo, Eva Onaindia. In 28th W*orkshop of the UK Planning and Scheduling Special Interest Group* (PlanSIG'10), Brescia (Italia), 2010.

7.2 Future works

In this Section, we present some future extensions concerning the more in-depth analysis of particular cases of the multi-agent reactive repair mechanism, such as, formalize some particular aspects of our Collaborative Repair, new proposals or just curiosity.

This Ph.D. dissertation has been mainly focused on the use of our MARPE model for recovering plan failures with repairing structures that are generating quickly at runtime. We formalized the main workflow of the MARPE model in Sections 5.3 and 5.4.1. However, we consider that some more particular aspects can be better formalized. For instance, during the plan execution of the collaborative solution, we can have two failing situations, one in which the requester agent fails and another where the helper agent fails (Section 5.4.4).

Despite the good results offered by our model, there are still many extensions that can significantly improve performance and efficiency. The multi-agent reactive repair mechanism consumes a high portion of the total time in the generation of new structures. The possibility of improving the structures' generation can be studied to do it incrementally, starting from an existing structure: the agent creates the search trees from a given partial state, which normally is the goal of the plan window and in many cases, it includes fluents that were used in previous structures. This improvement will increase the size of the structure that the agent can generate, and of course, it also will increase the number of encoded solutions in the tree.

Another possibility would be to incorporate learning techniques which allow storing search trees in a library of repairing structures. Thus, the agent generates the structure if the structure is not available in the library. Otherwise, the agent would only obtain the search space from the library instead of creating it from scratch.

Another line of research could be to integrate other multi-agent repair methods in our MARPE model, so we can ensure a more general model that could work with any repair method. However, so as we mentioned in Section 2, there is no other reactive collaborative repair method in the literature. Although, we found some deliberative multi-agent planning approach that we believe can be fixed to use more reactively.

Concerning the utilization of the MARPE model, we can also expect to apply the model to other real-world problems, such as the network traffic domains, or telescope control domains. As we show in Section 6, our model can work in different simulated planning domains of state of the art with pretty good results.

A final line of research focuses on the extension of the proposed multi-agent reactive repair method to support new functionalities, such as the numerical fluents. Working with numerical fluents is a fundamental characteristic of many problems with resources (e.g., the control of the fuel). Incorporating numerical variables in the search trees increases the cost of generating the structure. We may solve this problem by adjusting the regression model to consider this new feature.

Appendices

Appendix A

Planetary Mars scenario

A.1 Single-agent Mars scenario

A.1.1 PDDL3.1 Domain

```
(define (domain mars-scenario)
(:requirements :typing :equality :fluents)
(:types agent sample waypoint lander option - object
        rover - agent
        rock soil - sample)

(:constant  NONE - option)

(:predicates
            (link ?r - rover ?x - waypoint ?y - waypoint)
            (comm ?sa - sample ?x - waypoint)
            (analyze ?r - rover)
            (trans ?r - rover))

(:functions
            (loc ?x - (either lander rover)) - waypoint
            (have ?r - rover) - (either sample option)
```

```
             (locs ?sa - sample) - (either waypoint option))



(:action Navigate
 :parameters (?x - rover ?y - waypoint ?z - waypoint)
 :precondition  (and   (link ?x ?y ?z)
                       (= (loc ?x) ?y))
 :effect        (and   (assign (loc ?x) ?z)))

(:action Analyze
 :parameters (?x - rover ?sa - sample ?p - waypoint)
 :precondition  (and   (= (loc ?x) ?p)
                       (analyze ?x)
                       (= (locs ?sa) ?p))
 :effect        (and   (assign (have ?x) ?sa)
                       (assign (locs ?sa) NONE)))

(:action Communicate
 :parameters (?r - rover ?sa - sample ?l - lander ?x - waypoint ?y -
    waypoint)
 :precondition  (and   (= (loc ?r) ?x)
                       (= (loc ?l) ?y)
                       (= (have ?r) ?sa)
                       (trans ?r))
 :effect        (and   (comm ?sa ?x)
                       (assign (have ?r) NONE))))
```

A.1.2 List of variables

- `loc-B`: location of rover `B`.
- `loc-L`: location of the lander `L`.
- `locs-s`$_i$: location of an specific rock or soil sample `s`$_i$.

- `have-B`: that indicates what analysis the rover has; the value of this variable can be the results of a sample or NONE.
- `trans-B`: boolean variable that indicates whether there is a transmission device located in rover `B` or not.
- `analyze-B`: boolean variable that indicates whether rover `B` can analyze samples or not.
- `link-B`-w_i-w_j: boolean variable that indicates whether rover `B` has maps to travel w_i to w_j or not.
- `com`-s_i-w_j: boolean variable that indicates whether the communication of the analysis of a sample s_i has been successfully communicated from w_j or not.

A.1.3 PDDL initial state

The following piece of code shows the values of the variables that define the initial situation of the problem described in Section 3.3.1

```
(= (loc B) w2), (= (loc L) w2), (= (locs s1) w1), (= (locs s2) w3),
(= (locs r) w3), (analyze B), (trans B), ((= have B) NONE),
(link B w1 w2), (link B w1 w3), (link B w2 w1), (link B w2 w3),
(link B w3 w1), (link B w3 w2)
```

A.2 Multi-agent Mars scenario

A.2.1 PDDL3.1 domain

The following codes show the planning domain for each rover of the motivation example described in Section 5.1.

```
(define (domain mars-scenario-multi)
(:requirements :typing :equality :fluents)
(:types agent sample waypoint lander option - object
        rover - agent
        rock soil - sample)

(:constant  NONE - option)

(:predicates
            (link ?r - rover ?x - waypoint ?y - waypoint)
            (comm ?sa - sample ?x - waypoint)
            (analyze ?r - rover)
            (trans ?r - rover)
            (seek ?r - rover))

(:functions
            (loc ?x - (either lander rover)) - waypoint
            (have ?r - rover) - (either sample option)
            (locs ?sa - sample) - (either waypoint option))



(:action Seek
 :parameters (?r - rover ?sa - sample ?y - waypoint)
 :precondition  (and    (seek ?r)
                        (= (loc ?r) ?y))
 :effect        (and    (assign (locs ?sa) ?y)))

(:action Navigate
 :parameters (?r - rover ?y - waypoint ?z - waypoint)
 :precondition  (and    (link ?r ?y ?z)
                        (= (loc ?r) ?y))
 :effect        (and    (assign (loc ?r) ?z)))

(:action Analyze
 :parameters (?r - rover ?sa - sample ?p - waypoint)
 :precondition  (and    (= (loc ?r) ?p)
                        (analyze ?r)
                        (= (locs ?sa) ?p))
 :effect        (and
                        (assign (have ?r) ?sa)
                        (assign (locs ?sa) NONE)))

(:action Communicate
 :parameters (?r - rover ?sa - sample ?l - lander ?x - waypoint ?y -
    waypoint)
 :precondition  (and    (= (loc ?r) ?x)
                        (= (loc ?l) ?y)
                        (= (have ?r) ?sa)
                        (trans ?r))
 :effect        (and    (comm ?sa ?x)
                        (assign (have ?r) NONE)))

(:action Comm-rover
 :parameters    (?r - rover ?r2 - rover ?sa - sample ?w - waypoint)
 :precondition  (and    (= (loc ?r) ?w)
                        (= (have ?r) ?sa)
                        (trans ?r))
 :effect        (and
                        (assign (have ?r) ?sa))
                        (assign (have ?r2) ?sa))))
```

A.2.2 List of variables

The following codes show the definition of each variable used in the planning domain.

- `loc-X`: location of rover `X`.
- `loc-L`: location of the lander `L`.

- `locs-`s_i: location of an specific rock or soil sample s_i.
- `have-X`: that indicates what analysis the rover `X` has; the value of this variable can be the results of a sample or NONE.
- `trans-X`: boolean variable that indicates whether there is a transmission device located in rover `X` or not.
- `seek-X`: boolean variable that indicates whether rover `X` can seek for more samples or not.
- `analyze-X`: boolean variable that indicates whether rover `X` can analyze samples or not.
- `link-X-`w_i`-`w_j: boolean variable that indicates whether rover `X` has maps to travel w_i to w_j or not.
- `com-`s_i`-`w_j: boolean variable that indicates whether the communication of the analysis of a sample s_i has been successfully communicated from w_j or not.

A.2.3 PDDL3.1. problems

The following codes show the values of the planning problem for each rover of the motivation example described in Section 5.1.

Planning problem of rover `A`

```
(define (problem motivation-example)
(:domain mars-scenario-multi)
(:objects
        A B C            - rover
        r s1 s2          - sample
        L                - lander
        w1 w2 w3         - waypoint )
(:init
        ; location rover A
        (= (loc A) w2)

        ; location Lander
        (= (loc L) w2)

        ; location samples
        (= (locs s1) w1)
        (= (locs s2) w3)
        (= (locs r) w3))

        ; whether rover has capability to analyze
        (analyze A)

        ; whether rover has capability to communicate
        (trans A)

        ; whether rover has capability to seek
        (seek A)

        ; grid can traverse A
        (link A w1 w2) (link A w1 w3)
        (link A w2 w1) (link A w2 w3)
        (link A w3 w1) (link A w3 w2)

        (not (link A w1 w1)) (not (link A w2 w2)) (not (link A w3 w3))

        (= (have A) NONE)
        (= (have B) NONE)
        (= (have C) NONE)

        (not (comm r w1))  (not (comm r w2))  (not (comm r w3))
        (not (comm s1 w1)) (not (comm s1 w2)) (not (comm s1 w3))
        (not (comm s2 w1)) (not (comm s2 w2)) (not (comm s2 w3))
```

```
(:goal (and
        (comm r w3)
        (= (loc A) w2))))
```

Planning problem of rover B

```
(define (problem motivation-example)
(:domain mars-scenario-multi)
(:objects
        A B C           - rover
        r s1 s2         - sample
        L               - lander
        w1 w2 w3        - waypoint )
(:init
        ; location rover B
        (= (loc B) w2)

        ; location Lander
        (= (loc L) w2)

        ; location samples
        (= (locs s1) w1)
        (= (locs s2) w3)
        (= (locs r) w3))

        ; whether rover has capability to analyze
        (analyze B)

        ; whether rover has capability to communicate
        (trans B)

        ; whether rover has capability to seek
        (seek B)

        ; grid can traverse B
        (link B w1 w2) (link B w1 w3)
        (link B w2 w1) (link B w2 w3)
        (link B w3 w1) (link B w3 w2)

        (not (link B w1 w1)) (not (link B w2 w2)) (not (link B w3 w3))

        (= (have A) NONE)
        (= (have B) NONE)
        (= (have C) NONE)

        (not (comm r w1))  (not (comm r w2))  (not (comm r w3))
        (not (comm s1 w1)) (not (comm s1 w2)) (not (comm s1 w3))
        (not (comm s2 w1)) (not (comm s2 w2)) (not (comm s2 w3))

(:goal (and
        (comm S1 w1)
        (= (loc B) w2))))
```

Planning problem of rover C

```
(define (problem motivation-example)
(:domain mars-scenario-multi)
(:objects
        A B C           - rover
        r s1 s2         - sample
        L               - lander
        w1 w2 w3        - waypoint )
(:init
        ; location rover C
        (= (loc C) w2)

        ; location Lander
        (= (loc L) w2)

        ; location samples
        (= (locs s1) w1)
        (= (locs s2) w3)
        (= (locs r) w3))

        ; whether rover has capability to analyze
        (analyze C)

        ; whether rover has capability to communicate
        (trans C)

        ; whether rover has capability to seek
        (not (seek C))

        ; grid can traverse C
        (link C w1 w2) (link C w1 w3)
        (link C w2 w1) (not (link C w2 w3))
        (link C w3 w1) (link C w3 w2)

        (not (link C w1 w1)) (not (link C w2 w2)) (not (link C w3 w3))

        (= (have A) NONE)
        (= (have B) NONE)
        (= (have C) NONE)

        (not (comm r w1))  (not (comm r w2))  (not (comm r w3))
        (not (comm s1 w1)) (not (comm s1 w2)) (not (comm s1 w3))
        (not (comm s2 w1)) (not (comm s2 w2)) (not (comm s2 w3))

(:goal (and
        (comm s2 w3)
        (= (loc C) w2))))
```

A.2.4 Solution plans

The following codes show the initial solution plans for each rover of the motivation example described in Section 5.1.

plan of rover `A`

0 : (Navigate A w_2 w_3)

1 : (Analyze A r w_3)

2 : (Communicate A r L w_3 w_2)

3 : (Navigate A w_3 w_2)

plan of rover `B`

0 : (Navigate B w_2 w_1)

1 : (Analyze B s1 w_1)

2 : (Communicate B s1 L w_1 w_2)

3 : (Navigate B w_1 w_2)

plan of rover `C`

0 : (Navigate C w_2 w_1)

1 : (Navigate C w_1 w_3)

2 : (Analyze C s2 w_3)

3 : (Communicate C s2 L w_3 w_2)

4 : (Navigate C w_3 w_2)

Appendix B

Rovers domain

B.1 Single-agent `rovers`

This single-agent domain is a modification of the `rovers` domain of IPC of 2002 in the following aspects:

- We modeled the domain in PDDL3.1.
- We incorporated a set of new functionalities to promote more alternatives to repair plan failures.

The new functionalities are the following:

1. We modify the variable `have-B`: that indicates what the rover has; with the variable `have-B-sa`: that indicates the rover has an specify rock or soil sample; the value of this variable is also modified to have either the location where the sample was analyzed or NONE.
2. We endow rovers with reconnaissance abilities like, for instance, the operator (`Seek ?r ?sa ?w`) that allows a rover `?r` to seek more samples `?sa` in a waypoint `?w`.

3. We endow rovers with photography abilities like, for instance, the operator (Calibrate ?r ?i ?t ?w) that allows a rover ?r to calibrate the camera ?i in order to take a picture to the target ?t from the waypoint ?w or (Take_image ?r ?p ?t ?i ?m) that allows a rover ?r to take a picture to the target ?t using the camera ?i in high or low resolution mode ?m.

PDDL3.1 domain

```
(define (domain rover)
(:requirements :typing :equality :fluents)
(:types rover sample waypoint lander camera mode objective option - object
        rock soil - sample)

(:constants  NONE - option)

(:predicates
  (link ?r - rover ?x - waypoint ?y - waypoint)
  (comm ?sa - sample ?w - waypoint)
  (comm_image ?o - objective ?m - mode)
  (analyze ?r - rover ?sa - sample)
  (trans ?r - rover)
  (seek ?r - rover)
  (image ?r - rover)
  (supports ?c - camera ?m - mode)
  (have_image ?r - rover ?o - objective ?m - mode)
  (calibrated ?c - camera ?r - rover)
  (visible_from ?o - objective ?w - waypoint))

(:functions
  (loc ?x - (either lander rover)) - waypoint
  (have ?r - rover ?x - waypoint) - (either sample option)
  (locs ?sa - sample) - (either waypoint option)
  (calibration_target ?i - camera) - objective
  (on_board ?i - camera) - rover)

(:action Seek
 :parameters    (?r - rover ?sa - sample ?w - waypoint)
 :precondition (and     (= (locs ?sa) NONE)
                        (= (loc ?r) ?w)
                        (seek ?r))
 :effect       (and     (assign (locs ?sa) ?w)))

(:action Navigate
 :parameters (?x - rover ?y - waypoint ?z - waypoint)
 :precondition  (and    (link ?x ?y ?z)
                        (= (loc ?x) ?y))
 :effect        (and    (assign (loc ?x) ?z)))

(:action Analyze
 :parameters (?x - rover ?sa - sample ?p - waypoint)
 :precondition (and     (= (loc ?x) ?p)
                        (analyze ?x)
                        (= (locs ?sa) ?p))
 :effect       (and     (assign (have ?x ?sa) ?p)
                        (assign (locs ?sa) NONE)))

(:action Communicate
 :parameters (?r - rover ?sa - sample ?l - lander ?x - waypoint ?y -
    waypoint)
 :precondition  (and    (= (loc ?r) ?x)
                        (= (loc ?l) ?y)
                        (= (have ?r ?sa) ?x)
                        (trans ?r))
 :effect        (and    (comm ?sa ?x)
                        (assign (have ?r ?sa) NONE))))

(:action Calibrate
 :parameters    (?r - rover ?i - camera ?t - objective ?w - waypoint)
 :precondition (and     (imaging ?r)
                        (= (calibration_target ?i) ?t)
                        (= (loc ?r) ?w)
                        (visible_from ?t ?w)
                        (= (on_board ?i) ?r))
 :effect        (and    (calibrated ?i ?r))

(:action Take_image
 :parameters    (?r - rover ?p - waypoint ?t - objective ?i - camera ?m -
    mode)
 :precondition  (and    (calibrated ?i ?r)
                        (= (on_board ?i) ?r)
                        (image ?r)
                        (supports ?i ?m)
                        (visible_from ?t ?p)
                        (= (loc ?r) ?p))
 :effect        (and    (have_image ?r ?t ?m)
                        (not (calibrated ?i ?r))))

(:action Communicate_image
 :parameters   (?r - rover ?o - objective ?m - mode ?l - lander ?x - waypoint
     ?y - waypoint)
 :precondition  (and    (= (loc ?r) ?x)
                        (= (loc ?l) ?y)
                        (have_image ?r ?o ?m)
                        (trans ?r))
 :effect         (and   (comm_image ?o ?m)
                        (not (have_image ?r ?o ?m)))))
```

B.2 Multi-agent rovers

B.2.1 PDDL3.1 domain

```
(define (domain rover)
(:requirements :typing :equality :fluents)
(:types agent sample waypoint lander camera mode objective option - object
         rover - agent)

(:constants  NONE - option)

(:predicates
  (link ?r - rover ?x - waypoint ?y - waypoint)
  (comm ?sa - sample ?w - waypoint)
  (comm_image ?o - objective ?m - mode)
  (analyze ?r - rover ?sa - sample)
  (trans ?r - rover)
  (seek ?r - rover)
  (image ?r - rover)
  (trans_rover_to_rover ?r - rover)
  (supports ?c - camera ?m - mode)
  (have_image ?r - rover ?o - objective ?m - mode)
  (calibrated ?c - camera ?r - rover)
  (visible_from ?o - objective ?w - waypoint))

(:functions
  (loc ?x - (either lander rover)) - waypoint
  (have ?r - rover ?x - waypoint) - (either sample option)
  (locs ?sa - sample) - (either waypoint option)
  (calibration_target ?i - camera) - objective
  (on_board ?i - camera) - rover)

(:action Seek
 :parameters    (?r - rover ?sa - sample ?w - waypoint)
 :precondition (and      (= (locs ?sa) NONE)
                         (= (loc ?r) ?w)
```

```
                          (seek ?r))
 :effect         (and     (assign (locs ?sa) ?w)))

(:action Navigate
 :parameters (?x - rover ?y - waypoint ?z - waypoint)
 :precondition  (and     (link ?x ?y ?z)
                          (= (loc ?x) ?y))
 :effect         (and     (assign (loc ?x) ?z)))

(:action Analyze
 :parameters (?x - rover ?sa - sample ?p - waypoint)
 :precondition (and      (= (loc ?x) ?p)
                          (analyze ?x)
                          (= (locs ?sa) ?p))
 :effect        (and      (assign (have ?x ?sa) ?p)
                          (assign (locs ?sa) NONE)))

(:action Communicate
 :parameters (?r - rover ?sa - sample ?l - lander ?x - waypoint ?y -
    waypoint)
 :precondition  (and     (= (loc ?r) ?x)
                          (= (loc ?l) ?y)
                          (= (have ?r ?sa) ?x)
                          (trans ?r))
 :effect         (and     (comm ?sa ?x)
                          (assign (have ?r ?sa) NONE))))

(:action Calibrate
 :parameters    (?r - rover ?i - camera ?t - objective ?w - waypoint)
 :precondition (and      (imaging ?r)
                          (= (calibration_target ?i) ?t)
                          (= (loc ?r) ?w)
                          (visible_from ?t ?w)
                          (= (on_board ?i) ?r))
 :effect        (and      (calibrated ?i ?r))
```

```
(:action Take_image
 :parameters   (?r - rover ?p - waypoint ?t - objective ?i - camera ?m -
    mode)
 :precondition (and    (calibrated ?i ?r)
                       (= (on_board ?i) ?r)
                       (image ?r)
                       (supports ?i ?m)
                       (visible_from ?t ?p)
                       (= (loc ?r) ?p))
 :effect       (and    (have_image ?r ?t ?m)
                       (not (calibrated ?i ?r))))

(:action Communicate_image
 :parameters  (?r - rover ?o - objective ?m - mode ?l - lander ?x - waypoint
     ?y - waypoint)
 :precondition  (and   (= (loc ?r) ?x)
                       (= (loc ?l) ?y)
                       (have_image ?r ?o ?m)
                       (trans ?r))
 :effect        (and   (comm_image ?o ?m)
                       (not (have_image ?r ?o ?m))))

(:action Communicate_rover
 :parameters   (?r - rover ?r2 - rover ?sa - sample ?p - waypoint ?x -
    waypoint)
 :precondition (and    (= (loc ?r) ?x)
                       (trans_rover_to_rover ?r)
                       (= (have ?r ?sa) ?p))
 :effect       (and    (assign (have ?r ?sa) NONE)
                       (assign (have ?r2 ?sa) ?p)))

(:action Communicate_image_rover
 :parameters   (?r - rover ?r2 - rover ?o - objective ?m - mode ?x -
    waypoint)
:precondition (and     (= (loc ?r) ?x)
                       (trans_rover_to_rover ?r)
                       (have_image ?r ?o ?m))
:effect       (and     (not (have_image ?r ?o ?m))
                       (have_image ?r2 ?o ?m))))
```

B.2.2 Solution plans

B.2.2.1 problem one

plan of rover1

```
0.0: (calibrate rover1 camera0 objective0 waypoint2) [1.0]
1.0: (sample_soil rover1 rover1store waypoint2) [1.0]
2.0: (communicate_soil_data rover1 general waypoint2 waypoint2 waypoint3) [1.0]
3.0: (take_image rover1 waypoint2 objective0 camera0 low_res) [1.0]
4.0: (communicate_image_data rover1 general objective0 low_res waypoint2 waypoint3) [1.0]
```

plan of rover2

```
0.0: (navigate rover2 waypoint2 waypoint1) [1.0]
1.0: (sample_soil rover2 rover2store waypoint1) [1.0]
2.0: (navigate rover2 waypoint1 waypoint4) [1.0]
3.0: (sample_rock rover2 rover2store waypoint4) [1.0]
4.0: (communicate_soil_data rover2 general waypoint1 waypoint4 waypoint3) [1.0]
5.0: (communicate_rock_data rover2 general waypoint4 waypoint4 waypoint3) [1.0]
```

B.2.2.2 problem two

plan of rover1

```
0.0: (calibrate rover1 camera0 objective0 waypoint1) [1.0]
1.0: (navigate rover1 waypoint1 waypoint3) [1.0]
2.0: (sample_soil rover1 rover1store waypoint3) [1.0]
3.0: (take_image rover1 waypoint3 objective0 camera0 low_res) [1.0]
4.0: (navigate rover1 waypoint3 waypoint4) [1.0]
5.0: (communicate_soil_data rover1 general waypoint3 waypoint4 waypoint3) [1.0]
6.0: (communicate_image_data rover1 general objective0 low_res waypoint4 waypoint3) [1.0]
```

plan of rover2

```
0.0: (navigate rover2 waypoint1 waypoint2) [1.0]
1.0: (sample_soil rover2 rover2store waypoint2) [1.0]
2.0: (navigate rover2 waypoint2 waypoint4) [1.0]
3.0: (sample_rock rover2 rover2store waypoint4) [1.0]
4.0: (sample_soil rover2 rover2store waypoint4) [1.0]
5.0: (communicate_soil_data rover2 general waypoint2 waypoint4 waypoint3) [1.0]
6.0: (communicate_soil_data rover2 general waypoint4 waypoint4 waypoint3) [1.0]
7.0: (communicate_rock_data rover2 general waypoint4 waypoint4 waypoint3) [1.0]
```

B.2.2.3 problem three

plan of rover1

```
0.0: (calibrate rover1 camera0 objective0 waypoint4) [1.0]
1.0: (navigate rover1 waypoint4 waypoint3) [1.0]
2.0: (sample_rock rover1 rover1store waypoint3) [1.0]
3.0: (take_image rover1 waypoint3 objective0 camera0 low_res) [1.0]
4.0: (navigate rover1 waypoint3 waypoint5) [1.0]
5.0: (communicate_rock_data rover1 general waypoint3 waypoint5 waypoint3) [1.0]
6.0: (communicate_image_data rover1 general objective0 low_res waypoint5 waypoint3) [1.0]
```

```
plan of rover2
0.0: (sample_soil rover2 rover2store waypoint4) [1.0]
1.0: (navigate rover2 waypoint4 waypoint1) [1.0]
2.0: (sample_rock rover2 rover2store waypoint1) [1.0]
3.0: (navigate rover2 waypoint1 waypoint5) [1.0]
4.0: (sample_soil rover2 rover2store waypoint5) [1.0]
5.0: (communicate_soil_data rover2 general waypoint5 waypoint5 waypoint3) [1.0]
6.0: (communicate_rock_data rover2 general waypoint1 waypoint5 waypoint3) [1.0]
7.0: (communicate_soil_data rover2 general waypoint4 waypoint5 waypoint3) [1.0]
```

B.2.2.4 problem four

```
plan of rover1
0.0: (calibrate rover1 camera1 objective1 waypoint4) [1.0]
1.0: (sample_rock rover1 rover1store waypoint4) [1.0]
2.0: (navigate rover1 waypoint4 waypoint6) [1.0]
3.0: (communicate_rock_data rover1 general waypoint4 waypoint6 waypoint3) [1.0]
4.0: (sample_rock rover1 rover1store waypoint6) [1.0]
5.0: (communicate_rock_data rover1 general waypoint6 waypoint6 waypoint3) [1.0]
6.0: (take_image rover1 waypoint6 objective1 camera1 low_res) [1.0]
7.0: (communicate_image_data rover1 general objective1 low_res waypoint6 waypoint3) [1.0]
```

```
plan of rover2
0.0: (navigate rover2 waypoint4 waypoint3) [1.0]
1.0: (sample_soil rover2 rover2store waypoint3) [1.0]
2.0: (navigate rover2 waypoint3 waypoint1) [1.0]
3.0: (sample_rock rover2 rover2store waypoint1) [1.0]
4.0: (communicate_soil_data rover2 general waypoint3 waypoint1 waypoint3) [1.0]
5.0: (sample_soil rover2 rover2store waypoint1) [1.0]
6.0: (communicate_soil_data rover2 general waypoint1 waypoint1 waypoint3) [1.0]
7.0: (communicate_rock_data rover2 general waypoint1 waypoint1 waypoint3)
```

B.2.2.5 problem five

plan of rover1

```
0.0: (calibrate rover1 camera0 objective0 waypoint1)
1.0: (navigate rover1 waypoint1 waypoint6)
2.0: (sample_soil rover1 rover1store waypoint6)
3.0: (take_image rover1 waypoint6 objective0 camera0 low_res)
4.0: (communicate_image_data rover1 general objective0 low_res waypoint6 waypoint1)
5.0: (communicate_soil_data rover1 general waypoint6 waypoint6 waypoint1)
```

plan of rover2

```
0.0: (sample_rock rover2 rover2store waypoint4)
1.0: (navigate rover2 waypoint4 waypoint5)
2.0: (sample_rock rover2 rover2store waypoint5)
3.0: (communicate_rock_data rover2 general waypoint5 waypoint5 waypoint1)
4.0: (communicate_rock_data rover2 general waypoint4 waypoint5 waypoint1)
```

plan of rover3

```
0.0: (calibrate rover3 camera2 objective2 waypoint2)
1.0: (sample_rock rover3 rover3store waypoint2)
2.0: (communicate_rock_data rover3 general waypoint2 waypoint2 waypoint1)
3.0: (take_image rover3 waypoint2 objective2 camera2 low_res)
4.0: (communicate_image_data rover3 general objective2 low_res waypoint2 waypoint1)
```

B.2.2.6 problem six

plan of rover1

```
0.0: (calibrate rover1 camera0 objective0 waypoint1)
1.0: (navigate rover1 waypoint1 waypoint6)
2.0: (take_image rover1 waypoint6 objective0 camera0 low_res)
3.0: (communicate_image_data rover1 general objective0 low_res waypoint6 waypoint1)
4.0: (sample_soil rover1 rover1store waypoint6)
5.0: (communicate_soil_data rover1 general waypoint6 waypoint6 waypoint1)
```

plan of rover2

0.0: (sample_rock rover2 rover2store waypoint4)

1.0: (navigate rover2 waypoint4 waypoint5)

2.0: (sample_rock rover2 rover2store waypoint5)

3.0: (communicate_rock_data rover2 general waypoint5 waypoint5 waypoint1)

4.0: (communicate_rock_data rover2 general waypoint4 waypoint5 waypoint1)

plan of rover3

0.0: (calibrate rover3 camera2 objective2 waypoint2)

1.0: (take_image rover3 waypoint2 objective2 camera2 low_res)

2.0: (communicate_image_data rover3 general objective2 low_res waypoint2 waypoint1)

3.0: (sample_rock rover3 rover3store waypoint2)

4.0: (communicate_rock_data rover3 general waypoint2 waypoint2 waypoint1)

plan of rover4

0.0: (seek_rocks rover4 waypoint7)

1.0: (sample_rock rover4 rover4store waypoint7)

2.0: (navigate rover4 waypoint7 waypoint3)

3.0: (sample_soil rover4 rover4store waypoint3)

4.0: (navigate rover4 waypoint3 waypoint7)

5.0: (communicate_soil_data rover4 general waypoint3 waypoint7 waypoint1)

6.0: (communicate_rock_data rover4 general waypoint7 waypoint7 waypoint1)

Appendix C

Elevators domain

C.1 Single-agent `elevators`

PDDL3.1 domain

```
(define (domain elevators)
  (:requirements :typing)
  (:types elevator passenger floor status - object)

(:predicates
        (reachable-floor ?lift - elevator ?floor - floor)
        (above ?lift - elevator ?floor1 - floor ?floor2 - floor)
        (working-door ?lift - elevator ?n1 - floor))

(:constants YES NOT - option)

(:functions (lift-at ?lift - elevator) - floor
            (passenger-at ?person - passenger) - (either floor elevator)
            (working-lift ?lift - elevator) - option)

(:action move-up
  :parameters (?lift - elevator ?f1 - floor ?f2 - floor)
  :precondition (and (= (working-lift ?lift) YES)
                        (= (lift-at ?lift) ?f1)
                        (above ?lift ?f1 ?f2)
                        (reachable-floor ?lift ?f2))
  :effect         (and (assign (lift-at ?lift) ?f2)))

(:action move-down
  :parameters (?lift - elevator ?f1 - floor ?f2 - floor )
  :precondition (and (= (lift-at ?lift) ?f1)
                        (= (working-lift ?lift) YES)
                        (above ?lift ?f2 ?f1)
                        (reachable-floor ?lift ?f2))
  :effect         (and (assign (lift-at ?lift) ?f2)))

(:action board
  :parameters (?lift - elevator ?p - passenger ?f - floor)
  :precondition (and (working-door ?lift ?f)
                        (= (passenger-at ?p) ?f)
                        (= (lift-at ?lift) ?f))
  :effect         (and (assign (passenger-at ?p) ?lift)))

(:action leave
  :parameters (?lift - elevator ?p - passenger ?f - floor)
  :precondition (and (working-door ?lift ?f)
                        (= (passenger-at ?p) ?lift)
                        (= (lift-at ?lift) ?f))
  :effect         (and (assign (passenger-at ?p) ?f))))
```

C.2 Multi-agent `elevators`

C.2.1 PDDL3.1 domain

```
(define (domain elevators)
  (:requirements :typing)
  (:types agent passenger floor status - object
          elevator - agent)

(:predicates
        (reachable-floor ?lift - elevator ?floor - floor)
        (above ?lift - elevator ?floor1 - floor ?floor2 - floor)
        (working-door ?lift - elevator ?n1 - floor))

(:constants YES NOT - option)

(:functions (lift-at ?lift - elevator) - floor
            (passenger-at ?person - passenger) - (either floor elevator)
            (working-lift ?lift - elevator) - option)

(:action move-up
  :parameters (?lift - elevator ?f1 - floor ?f2 - floor)
  :precondition (and (= (working-lift ?lift) YES)
                     (= (lift-at ?lift) ?f1)
                     (above ?lift ?f1 ?f2)
                     (reachable-floor ?lift ?f2))
  :effect       (and (assign (lift-at ?lift) ?f2)))

(:action move-down
  :parameters (?lift - elevator ?f1 - floor ?f2 - floor )
  :precondition (and (= (lift-at ?lift) ?f1)
                     (= (working-lift ?lift) YES)
                     (above ?lift ?f2 ?f1)
                     (reachable-floor ?lift ?f2))
  :effect       (and (assign (lift-at ?lift) ?f2)))
(:action board
  :parameters (?lift - elevator ?p - passenger ?f - floor)
  :precondition (and (working-door ?lift ?f)
                     (= (passenger-at ?p) ?f)
                     (= (lift-at ?lift) ?f))
  :effect       (and (assign (passenger-at ?p) ?lift)))

(:action leave
  :parameters (?lift - elevator ?p - passenger ?f - floor)
  :precondition (and (working-door ?lift ?f)
                     (= (passenger-at ?p) ?lift)
                     (= (lift-at ?lift) ?f))
  :effect       (and (assign (passenger-at ?p) ?f))))
```

C.2.2 Solution plans

C.2.2.1 problem one

plan of elevator1

```
0.0: (move-up elevator1 floor2 floor3) [1.0]
1.0: (board elevator1 passenger2 floor3) [1.0]
2.0: (move-up elevator1 floor3 floor6) [1.0]
3.0: (leave elevator1 passenger2 floor6) [1.0]
4.0: (move-up elevator1 floor6 floor8) [1.0]
5.0: (board elevator1 passenger1 floor8) [1.0]
6.0: (move-down elevator1 floor8 floor4) [1.0]
7.0: (leave elevator1 passenger1 floor4) [1.0]
```

plan of elevator2

```
0.0: (move-down elevator2 floor4 floor2) [1.0]
1.0: (board elevator2 passenger3 floor2) [1.0]
2.0: (move-up elevator2 floor2 floor1) [1.0]
3.0: (leave elevator2 passenger3 floor1) [1.0]
```

C.2.2.2 problem two

```
plan of elevator1

0.0: (move-down elevator1 floor2 floor0) [1.0]
1.0: (board elevator1 passenger1 floor0) [1.0]
2.0: (move-up elevator1 floor0 floor4) [1.0]
3.0: (leave elevator1 passenger1 floor4) [1.0]
4.0: (board elevator1 passenger2 floor4) [1.0]
5.0: (move-up elevator1 floor4 floor6) [1.0]
6.0: (leave elevator1 passenger2 floor6) [1.0]
```

```
plan of elevator2

0.0: (move-down elevator2 floor4 floor2) [1.0]
1.0: (board elevator2 passenger3 floor2) [1.0]
2.0: (move-down elevator2 floor2 floor1) [1.0]
3.0: (leave elevator2 passenger3 floor1) [1.0]
```

C.2.2.3 problem three

```
plan of elevator1

0.0: (move-up elevator1 floor1 floor4) [1.0]
1.0: (board elevator1 passenger4 floor4) [1.0]
2.0: (move-up elevator1 floor4 floor5) [1.0]
3.0: (leave elevator1 passenger4 floor5) [1.0]
4.0: (move-up elevator1 floor5 floor6) [1.0]
5.0: (board elevator1 passenger3 floor6) [1.0]
6.0: (move-down elevator1 floor6 floor1) [1.0]
7.0: (leave elevator1 passenger3 floor1) [1.0]
```

plan of elevator2

```
0.0: (move-up elevator2 floor5 floor8) [1.0]
1.0: (board elevator2 passenger2 floor8) [1.0]
2.0: (move-down elevator2 floor8 floor1) [1.0]
3.0: (board elevator2 passenger1 floor1) [1.0]
4.0: (move-up elevator2 floor1 floor7) [1.0]
5.0: (leave elevator2 passenger2 floor7) [1.0]
6.0: (move-down elevator2 floor7 floor0) [1.0]
7.0: (leave elevator2 passenger1 floor0) [1.0]
```

C.2.2.4 problem four

plan of elevator1

```
0.0: (move-up elevator1 floor4 floor5) [1.0]
1.0: (board elevator1 passenger3 floor5) [1.0]
2.0: (move-up elevator1 floor5 floor6) [1.0]
3.0: (board elevator1 passenger4 floor6) [1.0]
4.0: (move-down elevator1 floor6 floor2) [1.0]
5.0: (leave elevator1 passenger4 floor2) [1.0]
6.0: (move-up elevator1 floor2 floor4) [1.0]
7.0: (leave elevator1 passenger3 floor4) [1.0]
```

plan of elevator2

```
0.0: (move-down elevator2 floor7 floor6) [1.0]
1.0: (board elevator2 passenger2 floor6) [1.0]
2.0: (board elevator2 passenger1 floor6) [1.0]
3.0: (move-down elevator2 floor6 floor1) [1.0]
4.0: (leave elevator2 passenger1 floor1) [1.0]
5.0: (move-up elevator2 floor1 floor3) [1.0]
6.0: (leave elevator2 passenger2 floor3) [1.0]
```

C.2.2.5 problem five

plan of elevator1

```
0.0: (board elevator1 passenger2 floor0) [1.0]
1.0: (move-up elevator1 floor0 floor2) [1.0]
2.0: (leave elevator1 passenger2 floor2) [1.0]
3.0: (move-up elevator1 floor2 floor4) [1.0]
4.0: (board elevator1 passenger1 floor4) [1.0]
5.0: (move-up elevator1 floor4 floor5) [1.0]
6.0: (leave elevator1 passenger1 floor5) [1.0]
```

plan of elevator2

```
0.0: (move-down elevator2 floor4 floor1) [1.0]
1.0: (board elevator2 passenger3 floor1) [1.0]
2.0: (move-up elevator2 floor1 floor8) [1.0]
3.0: (leave elevator2 passenger3 floor8) [1.0]
```

plan of elevator3

```
0.0: (board elevator3 passenger5 floor8) [1.0]
1.0: (move-down elevator3 floor8 floor1) [1.0]
2.0: (board elevator3 passenger4 floor1) [1.0]
3.0: (move-up elevator3 floor1 floor2) [1.0]
4.0: (leave elevator3 passenger4 floor2) [1.0]
5.0: (move-up elevator3 floor2 floor3) [1.0]
6.0: (leave elevator3 passenger5 floor3) [1.0]
```

C.2.2.6 problem six

plan of elevator1

```
0.0: (move-up elevator1 floor0 floor5) [1.0]
1.0: (board elevator1 passenger2 floor5) [1.0]
2.0: (move-down elevator1 floor5 floor3) [1.0]
3.0: (leave elevator1 passenger2 floor3) [1.0]
4.0: (board elevator1 passenger1 floor3) [1.0]
5.0: (move-up elevator1 floor3 floor6) [1.0]
6.0: (leave elevator1 passenger1 floor6) [1.0]
```

plan of elevator2

```
0.0: (move-down elevator2 floor7 floor3) [1.0]
1.0: (board elevator2 passenger3 floor3) [1.0]
2.0: (move-up elevator2 floor3 floor6) [1.0]
3.0: (leave elevator2 passenger3 floor6) [1.0]
```

plan of elevator3

```
0.0: (move-up elevator3 floor4 floor8) [1.0]
1.0: (board elevator3 passenger5 floor8) [1.0]
2.0: (board elevator3 passenger4 floor8) [1.0]
3.0: (move-down elevator3 floor8 floor0) [1.0]
4.0: (leave elevator3 passenger4 floor0) [1.0]
5.0: (move-up elevator3 floor0 floor7) [1.0]
6.0: (leave elevator3 passenger5 floor7) [1.0]
```

Appendix D

Transports domain

D.1 Single-agent `transport`

PDDL3.1 domain

```
(define (domain transport) (:requirements :typing :equality :fluents)
(:types agent location obj - object  truck)
(:predicates   (link ?truck - agent ?x ?y - location))

(:functions    (pos ?t - truck) - location
               (in ?o - obj) - (either location truck))

(:action load
 :parameters   (?truck - truck ?obj - obj ?loc - location)
 :precondition (and    (= (pos ?truck) ?loc) (= (in ?obj) ?loc))
 :effect       (assign  (in ?obj) ?truck))

(:action unload
 :parameters   (?truck - truck ?obj - obj ?loc - location)
 :precondition (and    (= (pos ?truck) ?loc) (= (in ?obj) ?truck))
 :effect       (assign  (in ?obj) ?loc))

(:action drive
 :parameters   (?truck - truck ?loc-from - location ?loc-to - location)
 :precondition (and    (= (pos ?truck) ?loc-from) (link ?truck ?loc-from ?
    loc-to))
 :effect       (assign  (pos ?truck) ?loc-to)))
```

D.2 Multi-agent `transport`

D.2.1 PDDL3.1 domain

domain of the truck agent

```
(define (domain transport) (:requirements :typing :equality :fluents)
(:types agent location obj - object  truck - agent)
(:predicates    (engine-operating ?t - truck)
                (hoist-operating ?t - truck)
                (link ?truck - agent ?x ?y - location))

(:functions     (pos ?t - truck) - location
                (in ?o - obj) - (either location truck))

(:action load
 :parameters    (?truck - truck ?obj - obj ?loc - location)
 :precondition  (and    (= (pos ?truck) ?loc) (= (in ?obj) ?loc) (hoist-
    operating ?truck))
 :effect        (assign (in ?obj) ?truck))

(:action unload-to-truck
 :parameters    (?truck1 - truck ?truck2 - truck ?obj - obj ?loc - location)
 :precondition  (and    (= (pos ?truck1) ?loc) (= (pos ?truck2) ?loc) (= (in
     ?obj) ?truck1))
 :effect        (assign (in ?obj) ?truck2))

(:action unload
 :parameters    (?truck - truck ?obj - obj ?loc - location)
 :precondition  (and  (= (pos ?truck) ?loc) (= (in ?obj) ?truck) (hoist-
    operating ?truck))
 :effect        (assign (in ?obj) ?loc))

(:action load-from-truck
 :parameters    (?truck1 - truck ?truck2 - truck ?obj - obj ?loc - location)
 :precondition  (and  (= (pos ?truck1) ?loc) (= (pos ?truck2) ?loc) (= (in
     ?obj) ?truck2))
 :effect        (assign (in ?obj) ?truck1))

(:action drive
 :parameters    (?truck - truck ?loc-from - location ?loc-to - location)
 :precondition  (and  (= (pos ?truck) ?loc-from) (link ?truck ?loc-from ?
    loc-to) (engine-operating ?truck))
 :effect        (assign (pos ?truck) ?loc-to)))
```

D.2.2 Solution plans

D.2.2.1 problem one

plan of truck1

0.0: (load truck1 package1 s0) [1.0]

1.0: (drive truck1 s0 s1) [1.0]

2.0: (unload truck1 package1 s1) [1.0]

plan of truck2

0.0: (load truck2 package2 s0) [1.0]

1.0: (drive truck2 s0 s1) [1.0]

2.0: (unload truck2 package2 s1) [1.0]

D.2.2.2 problem two

plan of truck1

```
0.0: (drive truck1 s0 s2) [1.0]
1.0: (load truck1 package1 s2) [1.0]
2.0: (drive truck1 s2 s1) [1.0]
3.0: (load truck1 package3 s1) [1.0]
4.0: (drive truck1 s1 s0) [1.0]
5.0: (unload truck1 package1 s0) [1.0]
6.0: (unload truck1 package3 s0) [1.0]
```

plan of truck2

```
0.0: (load truck2 package2 s1) [1.0]
1.0: (drive truck2 s1 s2) [1.0]
2.0: (unload truck2 package2 s2) [1.0]
```

D.2.2.3 problem three

plan of truck1

```
0.0: (drive truck1 s1 s0) [1.0]
1.0: (load truck1 package1 s0) [1.0]
2.0: (load truck1 package2 s0) [1.0]
3.0: (drive truck1 s0 s1) [1.0]
4.0: (unload truck1 package2 s1) [1.0]
5.0: (unload truck1 package1 s1) [1.0]
```

plan of truck2

```
0.0: (drive truck2 s2 s1) [1.0]
1.0: (load truck2 package3 s1) [1.0]
2.0: (drive truck2 s1 s2) [1.0]
3.0: (unload truck2 package3 s2) [1.0]
```

D.2.2.4 problem four

plan of truck1

```
0.0: (drive truck1 s1 s2) [1.0]
1.0: (load truck1 package1 s2) [1.0]
2.0: (drive truck1 s2 s1) [1.0]
3.0: (unload truck1 package1 s1) [1.0]
```

plan of truck2

```
0.0: (load truck2 package3 s0) [1.0]
1.0: (drive truck2 s0 s1) [1.0]
2.0: (load truck2 package4 s1) [1.0]
3.0: (drive truck2 s1 s2) [1.0]
4.0: (unload truck2 package3 s2) [1.0]
5.0: (drive truck2 s2 s0) [1.0]
6.0: (unload truck2 package4 s0) [1.0]
```

D.2.2.5 problem five

plan of truck1

```
0.0: (drive truck1 s1 s0) [1.0]
1.0: (load truck1 package2 s0) [1.0]
2.0: (load truck1 package1 s0) [1.0]
3.0: (drive truck1 s0 s1) [1.0]
4.0: (unload truck1 package1 s1) [1.0]
5.0: (unload truck1 package2 s1) [1.0]
```

plan of truck2

```
0.0: (drive truck2 s1 s2) [1.0]
1.0: (load truck2 package3 s2) [1.0]
2.0: (drive truck2 s2 s1) [1.0]
3.0: (unload truck2 package3 s1) [1.0]
```

plan of truck3

```
0.0: (drive truck3 s1 s2) [1.0]
1.0: (load truck3 package4 s2) [1.0]
2.0: (drive truck3 s2 s0) [1.0]
3.0: (unload truck3 package4 s0) [1.0]
```

D.2.2.6 problem six

plan of truck1

```
0.0: (drive truck1 s0 s1) [1.0]
1.0: (load truck1 package1 s1) [1.0]
2.0: (drive truck1 s1 s2) [1.0]
3.0: (unload truck1 package1 s2) [1.0]
```

plan of truck2

```
0.0: (drive truck2 s1 s0) [1.0]
1.0: (load truck2 package3 s0) [1.0]
2.0: (drive truck2 s0 s2) [1.0]
3.0: (unload truck2 package3 s2) [1.0]
```

plan of truck3

```
0.0: (load truck3 package5 s1) [1.0]
1.0: (load truck3 package4 s1) [1.0]
2.0: (load truck3 package2 s1) [1.0]
3.0: (drive truck3 s1 s0) [1.0]
4.0: (unload truck3 package5 s0) [1.0]
5.0: (drive truck3 s0 s2) [1.0]
6.0: (unload truck3 package4 s2) [1.0]
7.0: (unload truck3 package2 s2) [1.0]
```

Appendix E

Logistics domain

E.1 Single-agent `logistics`

The `logistics` domain is a significant problem in the industry of transportation [111] presented in the IPC of 2000. In this domain, there are several cities, each containing several locations, some of which are airports. There are also trucks, which can drive within a single city, and airplanes, which can fly between airports. The goal is to get some packages from various locations to various new locations. We used the same STRIPS version of the IPC with the only modification that we convert the domain and problem files from PDDL2.1 to PDDL3.1.

PDDL3.1 domain

```
(define (domain logistics) (:requirements :typing :equality :fluents)
(:types city place package vehicle - object  airport location - place  truck
     plane - vehicle)
(:predicates   (in-city ?loc - place ?city - city))
(:functions    (at ?a - vehicle) - place
               (in ?pkg - package) - (either place vehicle))

(:action load-truck
 :parameters   (?truck - truck ?pkg - package ?loc - place)
 :precondition (and (= (at ?truck) ?loc) (= (in ?pkg) ?loc))
 :effect       (and (assign (in ?pkg) ?truck)))

(:action unload-truck
 :parameters   (?truck - truck ?pkg - package ?loc - place)
 :precondition (and (= (at ?truck) ?loc) (= (in ?pkg) ?truck))
 :effect       (and (assign (in ?pkg) ?loc)))

(:action drive-truck
 :parameters   (?truck - truck ?loc-from - place ?loc-to - place ?city -
    city)
 :precondition (and (= (at ?truck) ?loc-from)(in-city ?loc-from ?city)(in-
    city ?loc-to ?city))
 :effect       (and (assign (at ?truck) ?loc-to)))

(:action load-plane
 :parameters  (?plane - plane ?pkg - package ?loc - place)
 :precondition(and (= (in ?pkg) ?loc) (= (at ?plane) ?loc))
 :effect      (and (assign (in ?pkg) ?plane)))

(:action unload-plane
 :parameters   (?plane - plane ?pkg - package ?loc - place)
 :precondition (and (= (in ?pkg) ?plane) (= (at ?plane) ?loc))
 :effect       (and (assign (in ?pkg) ?loc)))

(:action fly-plane
 :parameters   (?plane - plane ?loc-from - airport ?loc-to - airport)
 :precondition (and (= (at ?plane) ?loc-from))
 :effect       (and (assign (at ?plane) ?loc-to))))
```

Appendix F

Evaluating the Reactive Planner

F.1 Calculating the effective branching factor

Given the total number of nodes N and the maximum depth m of a tree $\mathcal{T}$, the effective branching factor is the branching factor that a uniform tree of depth m would have in order to contain $N+1$ nodes. Therefore, $N+1{=}1+b+b^2+\cdots+b^m$ [1]. A different representation for the same formula is shown in Equation F.1.1, which allows us to calculate the total number of nodes N in non-uniform search trees [81] using the concept of effective branching factor.

$$N = \frac{b^{m+1} - b}{b - 1} \tag{F.1.1}$$

Although b can not be easily written explicitly as a function of m and N, we can design a plot of b versus N with some values of m in order to find an equivalent formula that helps us to easily clear b. The plot, shown in Figure F.1, compares Equation F.1.1 (black line) with the formula to calculate the number of nodes N in a uniform tree, b^m (red line). As we can observe, Equation F.1.1 is an increasing function that almost has the same form of the formula b^m. The main difference is

[1]For simplicity, we will represent the effective branching factor with the same letter b of the branching factor.

that b^m is shifted to the left. Considering this difference and our purpose to find an approximate formula of the Equation F.1.1, we manually computed [2] the shift in the red curve (b^m) to approach Equation F.1.1 as much as possible. This results in the function $(b + 0.44)^m$, which can be graphically seen in Figure F.2 (blue line).

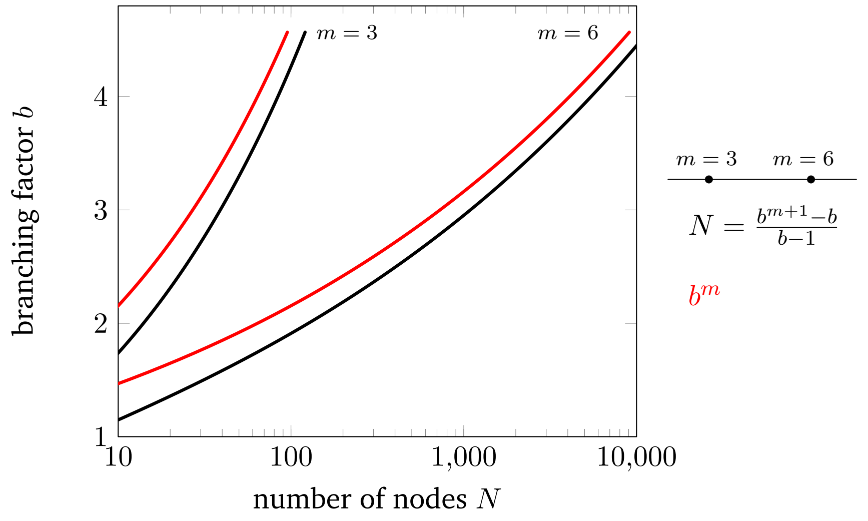

Figure F.1: Plot of the number of nodes N versus the branching factor b.

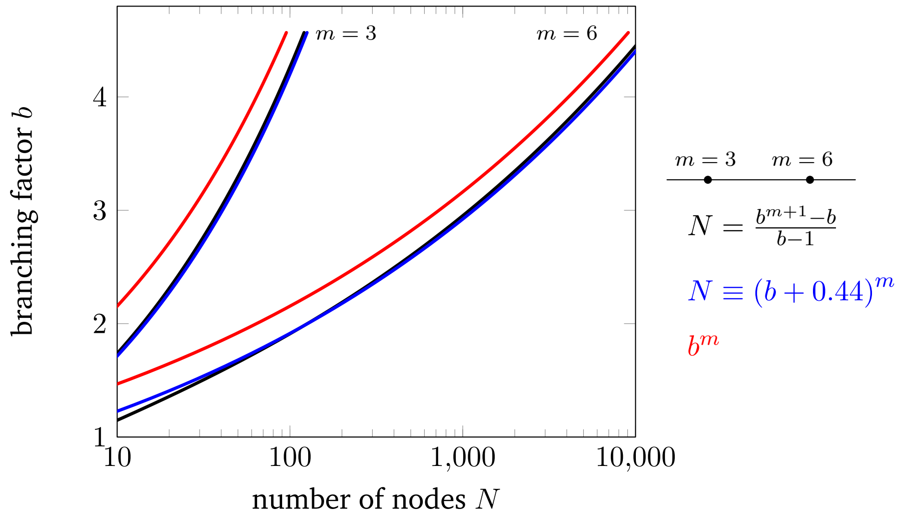

Figure F.2: Plot of N versus b with the approached function.

Two issues related to the previous formulas are worth mentioning here. First,

[2]We used the geometry package GeoGebra http://www.geogebra.org/

it is important to highlight that $(b+0.44)^m$ is an approximation of Equation 4.1.1, which can vary slightly depending on the values of m. Thereby, it is obvious that the function $(b+0.44)^m$ will not generate the real value of N but an approximate value. Secondly, the domain of values used for the branching factor b are positive values starting from 1 because a non-leaf node in a repairing structure will have at least one child.

F.2 Formulas to calculate the errors

F.2.1 Root Mean Square Error

The Root Mean Square Error (RMSE) [3] represents the sample **standard desviation** of the differences between the real values and the estimated values. It is computed for n different predictions as the square root of the mean of the squares of the differents between the estimated value and the real value. Values of zero indicates the estimated values are very close to the real values.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}} \tag{F.2.1}$$

F.2.2 Mean Absolute Error

The Mean Absolute Error (MAE) [4] is a quantity used to measure how close forecasts or predictions are to the eventual outcomes. It is computed as the average of the absolute errors, difference between the real time and the estimated time.

$$MAE = \frac{\sum_{i=1}^{n} \mid y_i - \hat{y}_i \mid}{n} \tag{F.2.2}$$

[3]https://en.wikipedia.org/wiki/Root-mean-square_deviation
[4]https://en.wikipedia.org/wiki/Mean_absolute_error

F.2.3 Mean Absolute Percentage Error

The Mean Absolute Percentage Error (MAPE) [5] is computed as the difference between the real time and the estimated time divide by the real time. The absolute value in this calculation is summed for every sample point and divided by the number of samples. Multiplying by 100 makes it a percentage error.

$$MAPE = \frac{\sum_{i=1}^{n} \mid \frac{y_i - \hat{y_i}}{y_i} \mid}{n} \tag{F.2.3}$$

[5]https://en.wikipedia.org/wiki/Mean_absolute_percentage_error

Appendix G

Conflict verification

In our MARPE model, we may choose to work with the verifying conflict process, in which case the repair solution of any agent j is considered valid if it does not produce any conflict with the plan window of the other external agents in the environment.

We declared the procedure `verify_conflicts` that checks if a given recovery solution is free of conflicts with the other plans that are been executing in the environment. Algorithm 5 shows the general workflout of the procedure `verify_conflicts`.

```
Input: $\Pi^j$, $\Delta$
                                                   external agent $k$
1: for $k \in \Delta$ do
2:     $j$ send $\Pi^j$ to $k$
3:                                                 $\Pi \leftarrow$ receive $\Pi^j$
4:                                                 if $conflictBetweenPlans(\Pi^k, \Pi^j) = false$ then
5:     $message \leftarrow$ receive response           send accept        ▷ Otherwise, send reject message
6:     if $message$ = reject then
7:         return $false$
8: end for
9: return $true$
```

Algorithm 5: Workflout of the procedure to verify conflicts

Let's assume that the repair process of the agent j found a solution plan Π' that solves the new planning task $\mathcal{P}'$ of the agent j such as we defined it at Section 5.3. The agent j will call the procedure of Algorithm 5, `verify_conflicts` (Π', Δ) where

Δ is a set of external agents. We define the function `verify_conflicts` as the procedure to check whether the solution Π' of the agent j is free of conflict with the other plans that are been executing in the environment.

The agent j sends each agent k of the set of external agents Δ the recovery solution (lines 1 and 2). The external agents receive the solution and accept the solution as valid if it is not conflicted with its current plan window (lines 3 - 6). Otherwise, the agent k reject the solution in which case the procedure `verify_conflicts` returns the value $false$ (lines 6 and 7).

We define the function $conflictBetweenPlans(\Pi^k, \Pi^j)$, line 4 of Algorithm 5, that given any two plans, Π^k and Π^j, checks whether a conflict between the plans exists or not by satisfying the following constraint. For every variable v published by agent k through the services, the agent j identifies pair of partial states $\{G_t^k, G_p^j\}$ $\subseteq \{\Pi^k \cup \Pi^j\}$ such as $v \subset \{ G_t^k, G_p^j\}$, and the partial states, G_t^k and G_p^j, must satisfy $\neg conflict(G_t^k, G_p^j)$. The procedure $conflictBetweenPlans(\Pi^k, \Pi^j)$ holds if it exists a partial state G_t of agent k that presents some mutex with any partial state G_p of the agent j in sequences of partial states of Π^k and Π^j, respectivelly.

We tested Algorithm 5 on a GNU/Linux Debian PC with an Intel 7 Core i7-3770 CPU @ 3.40GHz x 8, and 8 GB RAM; and in average it takes only 0.036 ms that represents 0,000036% of one execution cycle. This CPU time does not consider the communication delay between the agents in the environment. Moreover, in our experiments of Chapter 6 the agents are executing in the same PC, and thereby, the time of the communication between agents is imperceptible.

Bibliography

[1] Foundations for Intelligent Physical Agents. http://www.fipa.org/, April 2013. 20

[2] Competition of Distributed and Multiagent Planners, 2015. 165

[3] AgentScape. http://www.agentscape.org/about/, 2016. 21

[4] EVE. http://eve.almende.com/, 2016. 21

[5] JADE, Java Agent DEvelopment Framework. http://jade.tilab.com/, 2016. 21

[6] JASON. http://jason.sourceforge.net/wp/, 2016. 22

[7] Vidal Alcázar, César Guzmán, David Prior, Daniel Borrajo, Luis Castillo, and Eva Onaindia. PELEA: Planning, learning and execution architecture. In *28th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG10)*, pages 17–24, 2010. 14

[8] Pascal Aschwanden, Vijay Baskaran, Sara Bernardini, Chuck Fry, Maria Moreno, Nicola Muscettola, Chris Plaunt, David Rijsman, and Paul Tompkins. Model-Unified Planning and Execution for Distributed Autonomous System Control. In *AAAI Fall Symposium on Spacecraft Autonomy*, 2006. 62, 111

[9] C. Backstrom and B. Nebel. Complexity Results for SAS+ Planning. *Computational Intelligence*, 11(4):625–655, 1995. 35

[10] Ashis Gopal Banerjee and Satyandra K. Gupta. Research in Automated Planning and Control for Micromanipulation. *IEEE Transactions on Automation Science and Engineering*, 10(3):485–495, 2013. 1

[11] Mihai Barbuceanu and Mark S. Fox. COOL: A Language for Describing Coordination in Multi Agent Systems. In *First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pages 17–24, 1995. 20, 24

[12] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300, 1997. 34

[13] Blai Bonet, Héctor Palacios, and Hector Geffner. Automatic Derivation of Finite-State Machines for Behavior Control. In *Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010. 31

[14] Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah-Seghrouchni, Jorge J. Gómez-Sanz, João Leite, Gregory M. P. O'Hare, Alexander Pokahr, and Alessandro Ricci. A Survey of Programming Languages and Platforms for Multi-Agent Systems. *Informatica (Slovenia)*, 30(1):33–44, 2006. 19

[15] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996. 92

[16] Michael Brenner and Bernhard Nebel. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331, 2009. 1, 9

[17] B. Browning, J. Bruce, M. Bowling, and M. Veloso. STP: Skills, tactics and plays for multi-robot control in adversarial environments. *IEEE Journal of Control and Systems Engineering*, 219:33–52, 2005. 2, 9

[18] Joanna Bryson and Lynn Andrea Stein. Modularity and Design in Reactive Intelligence. *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 1115–1120, 2001. 2

[19] Steve Chien, Benjamin Cichy, Ashley Davies, Daniel Tran, Gregg Rabideau, Rebecca Castaño, Rob Sherwood, Dan Mandl, Stuart Frye, Seth Shulman, Jeremy Jones, and Sandy Grosvenor. An Autonomous Earth-Observing Sensorweb. *IEEE Intelligent Systems*, 20:16–24, 2005. 1

[20] Steve Chien, Russell Knight, Andre Stechert, Rob Sherwood, and Gregg Rabideau. Using Iterative Repair to Improve Responsiveness of Planning and Scheduling. In *Fifth Intl. Conference on Artificial Intelligence Planning and Scheduling*, pages 300–307, 2000. 14

[21] Joseph Y. J. Chow. Activity-Based Travel Scenario Analysis with Routing Problem Reoptimization. *Computer-Aided Civil and Infrastructure Engineering*, 29(2):91–106, 2014. 1

[22] Jacob Cohen and Jacob Cohen, editors. *Applied multiple regression/correlation analysis for the behavioral sciences*. L. Erlbaum Associates, 3rd ed edition, 2003. 87

[23] Andrew Coles, Maria Fox, Keith Halsey, Derek Long, and Amanda Smith. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence*, 173(1):1–44, 2009. 18

[24] Femke de Jonge and Nico Roos. Plan-execution health repair in a multi-agent system. *Workshop of the Planning and Scheduling Special Interest Group (PlanSIG)*, 2004. 39, 44

[25] Femke de Jonge, Nico Roos, and Cees Witteveen. Diagnosis of Multi-agent Plan Execution. In *Multiagent System Technologies, MATES 2006*, pages 86–97, 2006. 39

[26] Femke de Jonge, Nico Roos, and Cees Witteveen. Primary and secondary diagnosis of multi-agent plan execution. *Autonomous Agents and Multi-Agent Systems*, 18(2):267–294, 2009. 39

[27] M. de Weerdt and B. Clement. Introduction to planning in multiagent systems. *Multiagent and Grid Systems*, 5(4):345–355, 2009. 37

[28] Keith Decker. TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. *Foundations of Distributed Artificial Intelligence, Chapter 16*, pages 429–448, 1996. 24, 43

[29] Keith Decker and Victor R. Lesser. Generalizing the Partial Global Planning Algorithm. *International Journal of Cooperative Information Systems*, 2(2):319–346, 1992. 24

[30] M Bernardine Dias, Solange Lemai, and Nicola Muscettola. A real-time rover executive based on model-based reactive planning. *The 7th Intl. Symposium on Artificial Intelligence, Robotics and Automation in Space.*, 2003. 2

[31] Norman Richard Draper, Harry Smith, and Elizabeth Pownell. *Applied regression analysis*, volume 3. Wiley New York, 1966. 90

[32] Robert T. Effinger and Brian C. Williams. Dynamic Controllability of Temporally-flexible Reactive Programs. In *ICAPS*, pages 122–129. AAAI, 2009. 3

[33] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Towards Service Robots for Everyday Environments*, pages 49–64. Springer, 2012. 18

[34] Jacques Ferber. *Multi-agent systems - an introduction to distributed artificial intelligence*. Addison-Wesley-Longman, 1999. 19

[35] Dave Ferguson, Maxim Likhachev, and Anthony Stentz. A guide to heuristic-based path planning. In *International workshop on Planning Under Uncertainty for Autonomous Systems, International Conference on Automated Planning and Scheduling (ICAPS)*, pages 9–18, 2005. 33

[36] Richard E. Fikes, P. E. Hart, and Nils J. Nilsson. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 3:251–288, 1972. 17, 59

[37] Timothy W. Finin, Richard Fritzson, Donald P. McKay, and Robin McEntire. KQML As An Agent Communication Language. In *Third International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, 1994. 20

[38] Alberto Finzi, Felix Ingrand, and Nicola Muscettola. Model-based executive control through reactive planning for autonomous rovers. In *IEEE/RSJ Intl. Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 879–884, 2004. 37, 48, 62

[39] Lorenzo Flückiger and Hans Utz. Service Oriented Robotic Architecture for Space Robotics: Design, Testing, and Lessons Learned. *Journal of Field Robotics*, 31(1):176–191, 2014. 61

[40] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan Stability: Replanning versus Plan Repair. In *Conference on Automated Planning and Scheduling (ICAPS)*, pages 212–221, 2006. 17, 33, 102, 112

[41] Maria Fox and Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research (JAIR)*, 20:61–124, 2003. 54

[42] Maria Fox and Sylvie Thiébaux. Advances in automated plan generation. *Artificial Intelligence Journal*, 173(5-6):501–502, 2009. 1

[43] Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *ICML*, volume 96, pages 148–156, 1996. 93

[44] Christian Fritz and Sheila A. McIlraith. Monitoring Plan Optimality During Execution. In *Conference on Automated Planning and Scheduling (ICAPS)*, pages 144–151, 2007. 17, 60

[45] Antonio Garrido, César Guzmán, and Eva Onaindia. Anytime Plan-Adaptation for Continuous Planning. In *28th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG'10)*, Brescia (Italia), December 2010. 16, 17, 33

[46] E. Gat, M.G. Slack, D.P. Miller, and R.J. Firby. Path planning and execution monitoring for a planetary rover. In *IEEE International Conference on Robotics and Automation*, pages 20–25 vol.1, May 1990. 50

[47] Michael P. Georgeff and Amy L. Lansky. Reactive Reasoning and Planning. In *AAAI-87 Sixth National Conference on Artificial Intelligence*, pages 677–68, Seattle, WA (USA), July 1987. 2

[48] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning Through Stochastic Local Search and Temporal Action Graphs in LPG. *J. Artif. Intell. Res.(JAIR)*, 20:239–290, 2003. 18

[49] Alfonso E. Gerevini, Alessandro Saetti, and Ivan Serina. An approach to efficient planning with numerical fluents and multi-criteria plan quality. *Artificial Intelligence*, 172(8):899 – 944, 2008. 34, 37

[50] Alfonso E Gerevini and Ivan Serina. Efficient plan adaptation through replanning windows and heuristic goals. *Fundamenta Informaticae*, 102(3-4):287–323, 2010. 34, 37

[51] Bruce L Golden, Subramanian Raghavan, and Edward A Wasil. The vehicle routing problem: latest advances and new challenges. 43, 2008. 167

[52] César Guzmán, Vidal Alcázar, David Prior, Eva Onaindia, Daniel Borrajo, Juan Fdez-Olivares, and Ezequiel Quintero. PELEA: a Domain-Independent Architecture for Planning, Execution and Learning. In *ICAPS 6th Scheduling and Planning Applications woRKshop (SPARK)*, pages 38–45, 2012. 14, 28, 49

[53] César Guzmán, Pablo Castejon, Eva Onaindia, and Jeremy Frank. Robust Plan Execution in Multi-agent Environments. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014*, pages 384–391, 2014. 30, 134

[54] César Guzmán, Pablo Castejon, Eva Onaindia, and Jeremy Frank. Reactive execution for solving plan failures in planning control applications. *Integrated Computer-Aided Engineering*, 22(4):343–360, 2015. 61

[55] César Guzmán-Alvarez, Pablo Castejon, Eva Onaindia, and Jeremy Frank. Multi-agent Reactive Planning for Solving Plan Failures. In *Hybrid Artificial Intelligent Systems - 8th International Conference, HAIS 2013.*, volume 8073 of *Lecture Notes in Computer Science*, pages 530–539. Springer, 2013. 30, 61

[56] Michael R Hagerty and V Srinivasan. Comparing the predictive powers of alternative multiple regression models. *Psychometrika*, 56(1):77–85, 1991. 87

[57] M. Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246, 2006. 35

[58] Jörg Hoffmann and Ronen I Brafman. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6-7):507–541, 2006. 34

[59] Jörg Hoffmann et al. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *Journal of Artificial Intelligence Research (JAIR)*, 20:291–341, 2003. 35, 36

[60] Gunhee Kim and Woojin Chung. Tripodal Schematic Control Architecture for Integration of Multi-Functional Indoor Service Robots. *Industrial Electronics, IEEE Transactions on*, 53(5):1723–1736, Oct 2006. 12

[61] Antonín Komenda, Peter Novák, and Michal Pěchouček. Decentralized multi-agent plan repair in dynamic environments. In *11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1239–1240. International Foundation for Autonomous Agents and Multiagent Systems, 2012. 136

[62] Kurt Konolige. A gradient method for realtime robot control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2000, October 30 - Novemver 5, 2000, Takamatsu, Japan*, pages 639–646, 2000. 33

[63] Kazuhiro Kuwabara, Toru Ishida, and Nobuyasu Osato. AgenTalk: describing multiagent coordination protocols with inheritance. In *Seventh International Conference on Tools with Artificial Intelligence, ICTAI '95, Herndon, VA, USA, November 5-8, 1995*, pages 460–465, 1995. 20, 24

[64] Yoshiaki Kuwata, Alberto Elfes, Mark Maimone, Andrew Howard, Mihail Pivtoraiko, Thomas M Howard, and Adrian Stoica. Path Planning Challenges for Planetary Robots. In *IEEE Intl. Conference on Intelligent RObots Systems (IROS) 2nd Workshop*, pages 22–27, 2008. 3

[65] V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. Nagendra Prasad, A. Raja, R. Vincent, P. Xuan, and X. Q. Zhang. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):87–143, 2004. 2, 24, 43, 44, 127

[66] M. Lindstrom, A. Oreback, and H.I. Christensen. BERRA: a research architecture for service robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 4, pages 3278–3283 vol.4, 2000. 11

[67] Mark W Maimone, P Chris Leger, and Jeffrey J Biesiadecki. Overview of the mars exploration rovers' autonomous mobility and vision capabilities. In *IEEE Intl. Conference on Robotics and Automation (ICRA) Space Robotics Workshop*, 2007. 3

[68] Martín G. Marchetta and Raymundo Forradellas. An artificial intelligence planning approach to manufacturing feature recognition. *Computer-Aided Design*, 42(3):248–256, 2010. 1

[69] Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian P. Gerkey, and Kurt Konolige. The Office Marathon: Robust navigation in an indoor office environment. In *IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010*, pages 300–307, 2010. 33

[70] Mausam and Andrey Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. 31

[71] Conor McGann, Frederic Py, Kanna Rajan, John P. Ryan, and Richard Henthorn. Adaptive Control for Autonomous Underwater Vehicles. In *Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1319–1324, 2008. 13

[72] Conor McGann, Frederic Py, Kanna Rajan, Hans Thomas, Richard Henthorn, and Rob McEwen. A deliberative architecture for AUV control. In *IEEE Intl. Conference on Robotics and Automation (ICRA)*, pages 1049–1054, 2008. 2, 9

[73] Roberto Micalizio and Pietro Torasso. Team Cooperation for Plan Recovery in Multi-agent Systems. In *Multiagent System Technologies, 5th German Conference, MATES 2007, Leipzig, Germany, September 24-26, 2007*, pages 170–181, 2007. 38, 39, 44

[74] Roberto Micalizio and Pietro Torasso. Cooperative Monitoring to Diagnose Multiagent Plans. *J. Artif. Intell. Res. (JAIR)*, 51:1–70, 2014. 44

[75] Alfredo Milani and Valentina Poggioni. Planning in reactive environments. *Computational intelligence*, 23(4):439–463, 2007. 1

[76] Nicola Muscettola, Gregory A. Dorais, Chuck Fry, Richard Levinson, and Christian Plaunt. IDEA: Planning at the Core of Autonomous Reactive Agents. In *3rd Intl. NASA Workshop on Planning and Scheduling for Space*, 2002. 26

[77] Karen L Myers. CPEF: A continuous planning and execution framework. *AI Magazine*, 20(4):63, 1999. 14

[78] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. 54, 58, 64

[79] I.A.D. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin. CLARAty and challenges of developing interoperable robotic software. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, volume 3, pages 2428–2435 vol.3, Oct 2003. 27

[80] Nils J Nilsson. *Triangle tables: A proposal for a robot programming language*. SRI International. Artificial Intelligence Center, 1985. 60

[81] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. 89, 250

[82] Eva Onaindia, Oscar Sapena, Laura Sebastia, and Eliseo Marzal. SimPlanner: An Execution-Monitoring System for Replanning in Dynamic Worlds. In *Portuguese Conference on Artificial Intelligence*, pages 393–400. Springer, 2001. 35

[83] Jitu Patel, Michael C. Dorneich, David H. Mott, Ali Bahrami, and Cheryl Giammanco. Improving Coalition Planning by Making Plans Alive. *IEEE Intelligent Systems*, 28(1):17–25, 2013. 1

[84] Liam Pedersen, M Bualat, David Lees, David E Smith, David Korsmeyer, and R Washington. Integrated demonstration of instrument placement, robust execution and contingent planning. 2003. 34

[85] Stefan Poslad. Specifying protocols for multi-agent systems interaction. *TAAS*, 2(4), 2007. 20

[86] Anita Raja, Victor Lesser, and Thomas Wagner. Toward robust agent control in open environments. In *fourth international conference on Autonomous agents*, pages 84–91. ACM, 2000. 40

[87] Anand S. Rao. *AgentSpeak(L): BDI agents speak out in a logical computable language*, pages 42–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. 22

[88] Raymond Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*, volume 16. Cambridge Univ Press, 2001. 59

[89] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39(1):127–177, 2010. 6

[90] Silvia Richter, Matthias Westphal, and Malte Helmert. LAMA 2008 and 2011. *Intl. Planning Competition*, page 50, 2011. 18, 84, 102, 112

[91] Jussi Rintanen. Regression for Classical and Nondeterministic Planning. In *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008*, pages 568–572, 2008. 59

[92] Julio Rosenblatt. DAMN: a distributed architecture for mobile navigation. *J. Exp. Theor. Artif. Intell.*, 9(2-3):339–360, 1997. 10

[93] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995. 40

[94] Jessica A Samuels. The Mars Curiosity Rover Mission: remotely operating a science laboratory on the surface of another planet. In *17th Annual Intl. Space University Symposium, Strasbourg.* Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2013. 3

[95] Oscar Sapena and Eva Onaindía. A planning and monitoring system for dynamic environments. *Journal of Intelligent & Fuzzy Systems*, 12(3, 4):151–161, 2002. 35

[96] Oscar Sapena and Eva Onaindia. Planning in highly dynamic environments: an anytime approach for planning under time constraints. *Applied Intelligence*, 29(1):90–109, 2008. 36

[97] Oscar Sapena, Eva Onaindia, Antonio Garrido, and Marlene Arangú. A distributed CSP approach for collaborative planning systems. *Engineering Applications of Artificial Intelligence*, 21(5):698–709, 2008. 23

[98] Lin Padgham Sebastian Sardina. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, 2011. 2, 21

[99] Javier Sedano, Camelia Chira, José Villar, and Eva Ambel. An intelligent route management system for electric vehicle charging. *Integrated Computer-Aided Engineering*, 20(4):321–333, 2013. 1

[100] Sven Seuken and Shlomo Zilberstein. Formal models and algorithms for decentralized decision making under uncertainty. *Autonomous Agents and Multi-Agent Systems*, 17(2):190–250, 2008. 31

[101] R. Simmons, L.-J. Lin, and C. Fedor. Autonomous task control for mobile robots. In *5th IEEE International Symposium on Intelligent Control, vol. 2*, pages 663–668, 1990. 10, 36

[102] Marshall C Smart, BV Ratnakumar, LD Whitcanack, FJ Puglia, S Santee, and R Gitzendanner. Life verification of large capacity Yardney Li-ion cells and batteries in support of NASA missions. *Intl. Journal of Energy Research*, 34(2):116–132, 2010. 3

[103] David E Smith. Choosing Objectives in Over-Subscription Planning. In *Conference on Automated Planning and Scheduling (ICAPS)*, volume 4, page 393, 2004. 16

[104] Reid G Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, (12):1104–1113, 1980. 42

[105] Michal Stolba, Antonín Komenda, and Daniel L. Kovacs. *Competition of Distributed and Multi-Agent Planners (CoDMAP-15)*. AAAI, 2015. 38

[106] J. M. Such, A. Garcia-Fornes, A. Espinosa, and J. Bellver. Magentix2: a Privacy-enhancing Agent Platform. *Engineering Applications of Artificial Intelligence*, 2012. 28

[107] Jose M Such, Ana García-Fornes, Agustín Espinosa, and Joan Bellver. Magentix2: A privacy-enhancing agent platform. *Engineering Applications of Artificial Intelligence*, pages 96–109, 2012. 23, 38

[108] Katia Sycara, Massimo Paolucci, Martin van Velsen, and Joseph Giampapa. The RETSINA MAS Infrastructure. Technical report, The special joint issue of Autonomous Agents and MAS, Volume 7, Nos. 1 and 2, 2001. 2, 25

[109] Alejandro Torreño, Eva Onaindia, Antonín Komenda, and Michal Stolba.

Cooperative Multi-Agent Planning: A Survey. *ACM Computing Surveys*, 50(6):84:1–84:32, 2018. 38

[110] Alejandro Torreño, Eva Onaindia, and Oscar Sapena. FMAP: Distributed cooperative multi-agent planning. *Applied Intelligence*, 41(2):606–626, 2014. 38, 165

[111] Paolo Toth and Daniele Vigo. *The vehicle routing problem*, volume 9. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2002. 248

[112] Roman Van Der Krogt, Mathijs De Weerdt, and Cees Witteveen. A resource based framework for planning and replanning. *Web Intelligence and Agent Systems: An International Journal*, 1(3, 4):173–186, 2003. 40

[113] David E Wilkins and Karen L Myers. A Multiagent Planning Architecture., 1998. 2

[114] Brian C. Williams, Michel D. Ingham, S. H. Chung, and P. H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *IEEE: Modeling and Design of Embedded Software*, 91(1):212–237, 2003. 3

[115] Michael Wooldridge and Nicholas R Jennings. The cooperative problem-solving process. *Journal of Logic and Computation*, 9(4):563–592, 1999. 41, 44