Un composant logiciel pour la gestion et l'exécution de plan en robotique: Application aux systèmes multi-robots

Sylvain Joyeux

Thèse

en vue de l'obtention du Doctorat de l'Université de Toulouse
délivré par l'ISAE
(Domaine: Systèmes Informatiques Critiques)

présentée et soutenue par Sylvain JOYEUX
le 6 Décembre 2007

A Software Framework for Plan Management and Execution in Robotics :

Application to Multi-Robot Systems

Un composant logiciel pour la gestion et l'exécution de plan en robotique :

Application aux systèmes multi-robots

Thèse préparée au Laboratoire d'Analyse et d'Architecture des Systèmes
sous la direction de M. Simon Lacroix et M. Rachid Alami

Jury

M. Jacques MALENFANT	Rapporteur, Président
M. Michael BEETZ	Rapporteur
M^{me} Delphine DUFOURD	Examinatrice
M. Michel LEMAÎTRE	Examinateur
M. Simon LACROIX	Directeur de Thèse
M. Rachid ALAMI	Directeur de Thèse

Laboratoire d'Analyse et d'Architecture des Systèmes – UPR 8001

Contents

Résumé en français

List of Figures

Résumé

Dans les années 90, le problème de l'intégration des nombreuses fonctionalités nécessaires à l'autonomie de robots a donné naissance aux architectures robotiques, qui permettent aux différentes fonctions nécessaires aux robots autonomes de bien s'articuler entre elles. la perception, la décision et l'action. L'expérience dans ce domaine a montré les limites des différentes approches alors proposées. Récemment, de nouvelles architectures ont tenté de dépasser ces limites, principalement en unifiant la représentation du plan. Cette thèse propose à la fois un modèle de plan permettant de représenter les résultats de différents formalismes de décision, d'exécuter le plan qui en résulte, et de l'adapter en ligne. Ce modèle et le composant d'exécution et d'adaptation construit autour de lui ont été pensé dès l'origine pour le multi-robot: il s'agit de permettre l'exécution et l'adaptation de plans joints, c'est à dire de plans dans lesquels plusieurs robots coopèrent. Le composant logiciel construit durant cette thèse a de plus donné lieu à une validation expérimentale pour une coopération aéro-terrestre

Abstract

During the 90s, the integration of the many functionalities needed to make robot autonomous has given birth to robotic architectures, which allow cooperation between perception, decision and action in robotic systems. Experience with these architectures has shown that they suffer from limitations. More recently, new paradigms have appeared to tackle these limitations, based mainly on the idea that plan representation should be unified. This thesis contribution is a plan model which allows the integration of the result of different decision formalisms, to execute them and to adapt them online. Moreover, this model and the execution and adaptation component built around it have been designed with multi-robot in mind: it allows to build, execute and adapt joint plans, in which more than one robot are involved. The software component written during this thesis has been tested experimentally, in an aero-terrestrial cooperation scenario.

Remerciements

Ce manuscrit de thèse est le résultat de trois années de travail au LAAS/CNRS, à Toulouse. Je souhaite, par ces quelques mots, remercier les personnes qui ont rendu ces trois années riches à la fois humainement et scientifiquement.

Tout d'abord, merci à Malik Gallahb – directeur du laboratoire, Raja Chatila – chef du groupe de robotique puis directeur du laboratoire – et Rachid Alami – actuel chef de groupe – de m'avoir accueilli dans l'équipe de robotique du LAAS.

Merci à Simon Lacroix, mon directeur de thèse, pour avoir toujours cru en moi et avoir toujours espéré que de grandes choses ressortent de ce travail.

Merci à mes deux rapporteurs, Michael Beetz et Jacques Malenfant, pour avoir eu une lecture "différente" de ce manuscrit : leurs rapports en ont changé ma propre vision, à un moment où on a surtout "le nez dans le guidon".

Merci à ceux qui m'ont aidé à réaliser la partie expérimentale de cette thèse, partie à la fois la plus riche et la plus frustrante : Matthieu, Thierry et Arnaud au LAAS, Roger, Vincent et Alain au sein du projet Ressac de l'ONERA.

Merci à tous ceux qui ont rendu ces trois années aussi riches humainement. Les membres du groupe de thésards anonymes de 16h, ceux qui demandent si "ça va", et s'intéressent à la réponse, quand ils vous croisent à 20h (merci Patrick), ceux qui savent avoir le mot juste quand il faut. Merci à tous d'avoir supporté mes (nombreux) coups de gueule, parfois – je dois l'admettre – un peu répétitifs. Vous citer un par un serait un exercice certes intéressant, mais j'aurais trop peur d'en oublier un ... J'imagine que vous vous reconnaîtrez.

Enfin, merci à ceux et celles (et celle) qui m'ont fait oublier le travail dans des périodes où il devenait trop envahissant.

Introduction

The last ten years have seen tremendous progresses in the domain of autonomous robots: many technologies are now available which could allow to make autonomous robots go out of the research labs. Yet, no robots with high level of autonomy are out there, in the "real world": the martian rovers Spirit and Opportunity, while a remarkable success, have very few capabilities for decision-making. As we all tell when we present our research, autonomous robotic systems have many applications. But a lot of work is still needed to make them available out of research labs, because these applications require that humans simply tell the robot *what* to do, but no longer *how* to do it. An interesting exception is the Remote Agent experiment [15], where NASA let an autonomous system control a space probe for a few days. But it is still an exception, not the norm.

Recently, focus has been given on multi-robot systems: one single robot cannot achieve a lot, particularly when humans cannot be directly involved in the goal realizations, either because the task is too dangerous – like in fire-fighting or nuclear environments – or because we would like to achieve a 24/7 coverage – like in forest survey [32] or zone surveillance.

Focus

The work presented in this thesis contributes to that particular branch of the robotic research: multi-robot systems. It will also focus, for demonstration purposes, on an aero-terrestrial application.

In that context, our main focus is the definition and implementation of a *software architecture*, which is a widely, yet vaguely defined, concept. An interesting sentence about architecture is R. Simmons's: "architecture is the backbone of robotic systems" [64]. Architecture is, indeed, what should tie together all the software pieces a robot is made of. We will go forward in that way and say that architectures should *shape* how the developer (in our case, the people developing this software) should think about integrating their piece of work in the whole robot. Moreover, while conceptual architectures are sometimes defined separated to its actual implementation, we think it leads to define some very generic concepts which in no way helps in guiding actual developments. It is true that *concepts* are not tied to a particular implementation, but while a "conceptual" approach gives interesting insight about what components should be and how they should interact, it often does so at a level that does not help in actually getting things done.

Moreover, extensive work has already been done in defining modular architectures for basic functionalities, all of these architectures being more or less based on the notion of modules and

data flow, now common in software engineering. But to our knowledge very few architectures do try to integrate decision-making *as an external component* [2; 72; 27; 49]: while many systems have been deployed, in which decision making exists and is integrated, it is often done in an ad-hoc way. Since there is hardly one-fits-all automated planning tool, in particular in the case of multi-robot systems, we feel that generic integration of decision-making into a robotic architecture is a must-have. This is the leitmotiv of this work: one of our goals has been to provide a software framework in which it is possible to experiment in the integration of different decision-making tools, providing in the process a basis of reflexion for this integration.

Contributions

The main contributions of this thesis are:

- the definition of a generic software component for plan management, which defines the tools to build and to execute plans, to adapt them during execution, and to recover from various kind of errors.
- the definition of an architecture based on that component, which shows how it allows to integrate decision-making during the lifetime of the system.
- the adaptation of this architecture to the specificities of multi-robot systems: representation and manipulation of the notion of cooperation, limited communications.
- implementation of this system, and test both in simulation and in an experimental setup with two robots.

Outline

In the first chapter, we describe various approaches existing in the literature, analyzing their strength and drawbacks for both mono-robot and multi-robot systems. During this study of existing architectures, we try to show what are, in our opinion, the problems that architectures should solve. We finally describe our approach, how it relates to the state of the art, and introduce a supporting scenario which is used for illustration purposes in the remaining chapters.

In the second chapter, we describe what is a plan and how it is described in our system. We also show that plans from other plan models can be easily translated in our plan model.

In chapter 3, we show how these plans, once they are built, are executed by our system. We also show how non-nominal situations are handled.

Chapter 4 ties everything together by describing how our architecture allows to adapt plans during their execution, and how decision-making is handled during the system lifetime.

Then, chapter 5 describes issues specific to our implementation, and the related technical choices. It also describes how our system has been used to develop a controller for the two robots of our supporting scenario, and presents experimental results both in the field and in simulation.

Finally, chapter 6 summarizes our contributions, the obtained results and discusses the limitations of our system and how we would like to see it evolve.

1

Problem statement

The management of an autonomous robot – let alone a team of robots – is something challenging: it integrates lots of basic functionalities which should be put to work together to allow an autonomous behaviour in a complex, partially known, *dynamic* environment:

complex since it is the real world, either human-centric or natural. Because of this complexity, any model we can use to allow reasoning in our robot is bound to be imperfect – and thus the reasoning itself will be imperfect.

partially known even if the internal models the robot uses to reason were sufficiently close to reality, perception is an imperfect process and thus the knowledge the robot has of its environment is also imperfect. Autonomous robots have to reason on imperfect data and imperfect models.

dynamic the environment the robot evolves in changes over time. So, the estimate of the world state and the models the robot reason on at a particular time can become invalid while the robot is reasoning on them.

Since the 80's an area of robotic research has been to define *architectures*: principles and tools helping the process of managing the whole robotic system, acknowledging the fact that a robot must continuously adapt. Architecture deals with the integration of the multiple functionalities needed for the robot to work: it is the "backbone of robotic systems" [64]. Reflexions on architecture have given birth to many software systems which apply different approaches to build the software system of an autonomous robot.

This chapter gives an overview of the existing approaches for single robot and multi-robot control, showing how the integration of decision-making is integrated in existing systems, and the limits of the existing approaches. It also describes the problems specific to multi-robot systems and how these problems are handled by currently existing architectures. We will describe, based

on that reflexion, how our system is designed and a supporting scenario which will provide the examples needed in this dissertation, and show the viability of our approach.

First, how decision-making has been integrated in a single robot is presented (section 1.1). Then section 1.2 discusses the specific requirements of multi-robot systems and what approaches have been developed in that regard. Section 1.3 describes the approach we developed in order to handle the problems presented in the first two sections. Finally, section 1.4 describes the experimental scenario which allowed us to validate our system.

1.1 Decision-making in single robotic systems

The integration of decision-making in autonomous robots remains a challenge. As we will see, the complexity of decision-making algorithms impose to split the decision problems into smaller problems. The problem the architecture designer has then to face is to do that in a sound way: how to make the different decision-making components cooperate nicely ?

First, an overview of the two main architecture paradigms is presented. Then, we discuss the structure of decision making in robotic systems, how the separation of decision making into small components can impact the overall system behaviour and how some architectures mitigate these impacts. Finally, we give an overview of the notions of error and error handling, which is central in the problem of system dependability, and how this problem is handled in current architectures.

1.1.1 The main approaches in mono-robot architectures

Traditionally, robotic architectures are classified in two main paradigms: the behaviour based architectures and the hybrid architectures. We took in this section some of the arguments and ideas presented in [35], which provides – in our opinion – a good historical and conceptual introduction to classical robot architectures.

In behaviour-based systems the robot control is implemented through a particular composition of simple behaviours. Brooks [19] basic idea, which is at the origin of this concept, was that the classical sense-plan-act loop of robot control could be reduced into a sense-act – removing the need for *explicit* representation of the robot state and an ahead of time reasoning about its future. From this original idea, each specific architecture defined its own concept of behaviour, and the way to compose them. Arkin's motor schemas [7], for instance are force-field generation procedures defining specific motion behaviours (for instance, *avoid-static-obstacles* or *move-ahead*). These specific behaviours are then composed by a weighted average of the force fields.

Layered architectures [2; 35; 65] – also called hybrid architectures – integrated the "plan" step on top of a behaviour-like system, to use classical AI techniques like task planning. As Bonasso et al. [16] put it:

> "Others sought to integrate traditional reasoning systems with the reactive style, a kind of "plan-sense-act approach". What emerged was a software control architecture

> with a lowest layer of reactivity, a topmost layer of traditional AI planning, and a middle layer which transformed the state-space representation of plans into the continuous actions of the robot.“

For this reason, these architectures are also called 3-layer architectures since they are more or less all based on the three-layers depicted above. It is interesting to see that “classical” behaviour architectures have integrated reconfiguration systems, based either on behaviours themselves [53; 54] or on other paradigms. AuRa [8], for instance, has a finite-state automata (FSA) which chooses the right configuration of motor schemas for the current situation the robot is in. Arkin et al. have extended this FSA-based approach to allow non-specialists to specify and monitor the robot missions, especially in military contexts [6]. This way of specifying the robot controller has been successful both in the field and in simulation for many applications [1].

In this thesis, we adopted the point of view of hybrid architectures: behaviours and ahead-of-time decision-making are complementary approaches, and architectures should support their integration. The next section therefore focuses on the structure of decision-making in robotic architectures: how decision is integrated in the classical architectures presented above, what are their limitations and what attempts have been made to address these limitations in modern architectures.

1.1.2 Where does decision take place ?

What is decision making in robotic architectures is a difficult question, and we do not pretend to give a definitive answer here. We only try to give, through examples, a feeling of the following three categories:

- the components which are definitely taking decisions.
- the components which are definitely *not* taking decisions.
- the grey area in which it is difficult to choose.

Low-level behaviours like “go forward” or “turn left” are obviously *not* decision-making components. Another example is a module to control motors, or a pilot module for an UAV. It does not *choose* anything: it has a very short-term order (for instance the desired path the robot should follow) and the role of these components are to stick to this order regardless of external influence (wind for the UAV for instance).

Action planning deals with deciding the future actions of the robot, given its goals, its current state, a model of its environment and a model of how it can change its own state in this environment. Action planning is definitely in the domain of decision making: it chooses, in the set of actions the robot can perform, the ones that are the most likely to make it reach its goal.

In the gray area, there is *path planning*, or – more generally – all processes which involve choosing the place the robot should move (for instance, a *perception planner* which tells where perception should be done). This kind of planning deals with choosing the right path for

[1] the interested reader can look at GeorgiaTech Mobile Robots' lab webpage at `http://www-static.cc.gatech.edu/ai/robot-lab`

the robot, given a *goal*. One should argue that given the environment cost function and the geometrical model of the robot, there is only one optimal path. To that, we can answer that, most of the time, a whole class of paths are equivalent from the robot mission point of view and that most of the planning algorithms choose a single path in all the possible equivalent ones. Path planning modules therefore do choose a path among the possible ones: they make a decision which constrains the whole robot.

This can be seen as a definition of decision-making: any component which *chooses* one of the possible, "sensible" futures of the robot do take a decision. As such, they are all *planners*: they do a projection of the robot into its possible futures and choose one (or many) among them. As we will see in the next section, it is important to understand the interaction between the different decision-making components – and to allow them to have rich interactions – as these interactions are critical for the overall robot performance.

1.1.3 The effects of knowledge separation

The example of path planning also illustrates well the problem of *model inconsistency* which appears when more than one component take decisions. Let's consider a robot which has an action to perform at a given location L. The classical architectural solution is for a "high level" action planner to select a movement towards L, and then let the functional layer plan it and execute the resulting path. The problem is on what criteria the path planning algorithm chooses the "right" path ? It is very easy to see that the method chosen is critical for the overall robot performance: let's consider a "good" path planning algorithm, which is able to make the robot reach its goal. Now, let's consider the two following situations:

- a "minor" goal is given to the robot during its movement, either because an operator has chosen to do so or because of its interactions with other robots. The chosen path can make that minor goal too costly to reach. If the path *choice* had taken the *possible* actions of the robot into account, then the right path would have been chosen. However, this would require to make the path planner have the knowledge of the robot action model.
- let's consider a more problematic case: the robot interacts with other robots and all the robots share their model of the environment. Path planning would have, in this case, to choose between paths that are *known* to be good, and paths that are partially unknown to it, but for which another robot accepts to perform a perception. Choosing the right path then becomes a process of cooperation between multiple robots, taking both the path planning process *per se* and the action planning processes into account.

One solution is to have a single planner which takes both sides into account [39]. However, this kind of inter-mixed geometric/symbolic planning is very expensive from a complexity point of view. It appears that keeping path planning and action planning separate is an order of magnitude more efficient from a computational complexity point of view.

However, *if the action planning process is able to reason about the path*, then another solution would be available in-between these two: (i) path planning generates a *set* of reasonable paths

and (ii) action planning then selects the "right" path based on its own action model. In the multi-robot example above, this would require to represent the dependencies between the execution of each generated paths and the perceptions of regions of interest.

One would therefore need to represent, in the action plan, the paths generated by the path planning tool, along with the dependencies between these paths and other actions. In both cases, the path planning algorithm would *still* be a decision making process: it would choose what is "reasonable" and what is not, but in a way which does not over-constrain the rest of decision making.

1.1.4 Towards unified representations

This problem also appears between action planners: in architectures where action planning is split into hierarchies – like the decision layers advocated by [2; 1] – the lower layers have a very partial knowledge of the plans of the upper layers: their knowledge is limited to the actions that should be executed right now. In these schemes, there is not the possibility to choose, in the lower layers, the optimal actions *given the future plan of the upper layer*. The lower layers can therefore over-constrain the upper layer.

Moreover, in the case of action planning, there is another problem to consider: layers may not share the same action model. This is known as the hierarchy of abstraction: the upper levels manipulate simpler models than lower layers. This has three consequences:

- the boundaries between the robot actions are defined beforehand. Therefore, one can not change the planner dynamically according to the situation. This is a problem since there is no (or at least, not yet) a *universal* planner: one must have the ability to choose the right planner for the job at hand.
- one can not check that the overall plan is coherent, since one does not have a representation of this overall plan.
- it is possible that the plan of the upper layer is in conflict with the plan of the lower layer: when executed, one of the two execution components detects a violation of its own model and make the action fail.

These problems have already been addressed at least partially in different ways by three systems: the Coupled Layer Architecture for Robotic Autonomy (CLARAty) [72; 73; 52], the IDEA architecture [49; 29] and the Concurrent Reactive Plans [12].

CLARAty addresses the problems outlined above by coupling a Casper planner [20] with a TDL execution layer [63] through a new component: the CLEaR System [21]. The goal of this component is to offer a view of the overall plan, and to check that this plan is *globally* coherent. Future goals for this system are the integration of more reasoning tools – as for instance the Casper scheduling engine – directly into CLEaR. For now, the only problem with the implementation of Claraty is that the execution is done by TDL, not by CLEaR itself: so, the global plan is not the executed one.

IDEA has chosen a completely different approach: the core principle in IDEA is that there are multiple planning engines, but that they must (i) share the same model and (ii) cooperate

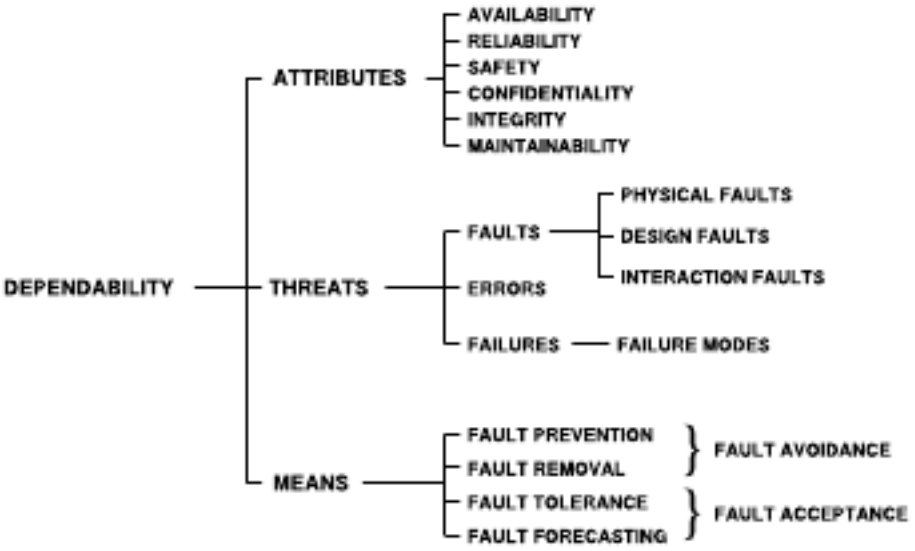


Figure 1.1: Dependability tree

so that the system guarantees a global consistency of the global plan. To achieve that, the designer of IDEA added the constraint that all agents must share the same planning model. IDEA can therefore only integrate tools which reason directly on this planning model, which is a very strong constraint. Nonetheless, the IDEA agent is a great achievement: it has proven that controlling robots by using only planning is possible. The same approach has successfully been used previously during the Remote Agent Experiment [41; 15], of which IDEA is a successor.

The Concurrent Reactive Plans system is also based on a plan-based control of robotic agents in which the plan is dynamically adapted to its changing environment. The main focus of this work is to first define a generic executable plan model and then develop transformation planning: given a plan and a change of situation, how to specify changes to the plan to make it compatible with the new situation. It is a very interesting work from our point of view because it shows that, given a rich enough plan model which can be different to the models used by the plan generation tool, one can (i) define a generic tool to adapt the plans in a sound manner and (ii) develop global plan analysis tools like plan-based state prediction [14].

1.1.5 Error representation and management

A good introduction to the problem of dependability and the underlying definitions (Fig. 1.1) can be found in [9]. [46] – from which comes the following definitions – applies this concepts to the robotic domain.

In the domain of dependability, one defines a *failure* as an event that occurs when the delivered service deviates from correct service. An *error* is that part of the system state which can cause a failure. An error is detected if its presence is indicated by an error message A *fault* is the adjudged or hypothesized cause of an error.

The means to make a system dependable are then regrouped in four concepts: (a) fault prevention, that is how to prevent the occurrence or introduction of faults, (b) fault removal, that is how to reduce the number or severity of faults, (c) fault tolerance, that is how to deliver correct service in the presence of faults, and (d) fault forecasting, that is how to estimate the present number, the future incidence, and the likely consequences of faults.

On one hand, in all the architectures we already mentioned but IDEA, the domain of *fault prevention* is taken by planning: the plan represents reactions to nominal and non-nominal situations, allowing a plan-based reaction in the presence of errors. Note that planners themselves are subject to faults, and even if the planning engine was perfect, it is possible to have error in models [46].

On the other hand, fault removal is traditionally handled by the tools designed to develop the intermediate layer of three-layer systems. Outside the IDEA approach, the traditional way to build that layer has been to develop languages or software libraries in which the robotic engineer writes procedures. These procedures are supposed to handle the details of translating the high-levels models manipulated by the planners into the orders which can be sent to the functional layer. They also have a role of fault removal: they handle some of the errors in the system, to reduce the complexity of the system from the planner point of view – which has then to handle only the remaining errors.

TDL [63], ESL [34], PRS [40] are examples of tools designed to help handling this process. They are all more or less based on the notion of context-based reactive refinement of the high level plan into low-level orders. The layers written in these systems are often called *supervision layers* as they send orders to the functional layer and supervise the correct execution of these orders.

The three tools mentioned above have in common that they are "not [designed] to serve as a representation for automated reasoning or formal analysis (although nothing precludes its use for these purposes)" [34]. Recently, systems have been designed to take into account the problem that procedures written in these systems are written by humans, and as such could contain faults. Without the help of automated formal analysis it is impossible to *prove* that the procedures would behave well. In robotic systems, it can have possibly dramatic consequences in the physical world.

Three main approaches exist. The first approach is to develop a provably-right supervision layer, for instance through Petri nets [10], finite-state automata [71], synchronous languages [50]. The second approach is the use of a "safety bag": continue to use supervision tools not based on models, but insert a thin model-based layer which checks that the orders sent by this supervision layer are acceptable [11; 58]. The third approach is – of course – the IDEA approach: to develop a whole model-based fault-tolerant architecture like [74].

The problem with the first two approaches are the limits of *error representation*: for these systems to be able to handle inconsistencies in the orders sent by upper layers and error reports coming from the lower layers, they must be able to represent and reason about them. The experience of the Request and Resource Checker (R^2C [58]) at LAAS showed that, since the PRS-based supervision layer was unable to reason about the errors detected by the safety bag,

the system as a whole was sometimes unable to properly handle that error: the supervision system was unable to differentiate between "the task failed to start" and "the safety bag did not allow to start it" which are two different things. This is yet another example of the limits of model separation: the models manipulated by the two layers are too different for them to interact properly.

1.1.6 Conclusion

In this section, we presented the most critical problems in layered decision-making: how separation of knowledge and model inconsistencies can impact a robotic system as a whole. Because of these observations, we decided to base our approach on the notion of a single *plan management* component: a component which gathers the results of all the decision making processes into a single central plan, and which executes that plan. Having this central place with the representation of the whole system plan should allow to develop online plan verification, conflict handling, ... which benefit greatly of a global representation of the robot plan.

The next section focuses on the problem of multi-robot in today's architectures: how multi-robot imposes new requirements on the robot architectures.

1.2 Multi-robot systems

Multi-robot systems induced multiple interesting problems when compared to the management of a single robot:

- the need for new planning models in which the presence of multiple, possibly heterogeneous, agents is taken into account: they need to represent the fact that all actions can not be performed by all robots.
- the need for new execution frameworks which take into account the fact that multiple robot are loosely coupled when compared to what is required by the management of a single robot.
- they must operate under limited communications: the robots may not have the ability to communicate at all times, and even when they do, the communication bandwidth is limited when compared to what's available in a single computer system.

Behavioural approaches [54; 47] exist for multi-robot systems: they are based on exactly the same principles that in mono-robot and have shown interesting results. Nonetheless, for the same reasons than outlined in the first section, we will not focus our attention on these architectures. Instead, our focus is planning in multi-robot systems.

1.2.1 Decision in multi-robot systems

When it comes to decision-making for multi-robot systems, one essential question is the *location* of the decision making process: decision-making can be either decentralized – each robot takes its own decisions – or centralized – one robot does all the decision making for all the other robots.

The interest of the first approach is robustness: to operate, the multi-robot system does not depend on one single decision-making system. The second approach is nonetheless interesting since the decision making process is not limited anymore by the communication problem: time is no more spent on communication, and the planner can manipulate more information about each robot.

The first difference between decision making for a single robot and for multi-robot systems is that one should decide and represent *who* is doing *what*. *Task allocation* is the most simple instance of that process: given a set of activities to realize, decide what robot will do what activity.

The Contract Net Protocol (CNP) is a very simple, yet very effective, protocol which has been designed to solve this problem [67]: a manager robot presents a set of tasks to be allocated, and the other robots bid for the tasks, based on their own capabilities and their other activities. The highest bidder is allocated the task and has then a contract with the bidding robot. The original protocol is a very effective one when the tasks are not coupled (i.e. there is never the constraint that two tasks must be allocated to the same robot): it is simple and completely decentralized. Extensions to the original protocol are the auction of task groups, the ability for robots to form groups, the group being the bidding entity. These approaches are now called "market based" because of their auction structure, and numerous works exist in that area [18; 36; 25; 44]. [37] presents and compares a few variants of CNP. It also compares them against behaviour-based approaches to task allocation.

The main limitation of the task allocation approach is that it models the robots as individual entities, and do not represent – for instance – that they have a common goal. Tambe et al. have centered their work on the representation of teams [69; 61; 59; 55]. In the context of these teams, the task allocation problem is replaced by the *role allocation problem*: in the team, the robot has no more a single activity but instead it is supposed to stick to solve a certain part of the problem. Another way to put it is that, when a robot is assigned a role, it accepts to realize a certain set of tasks, which are needed to achieve the team goal.

The second difference between decision making for a single robot and for multi-robot systems is that the existence of cooperation between robots create new ways to optimize the multi-robot system as a whole. For instance, the action of one robot can indirectly impact the actions of another: the possibilities of "soft" or "hard" interactions between each robot plans are much greater. For decentralized decision-making processes, it calls for new schemes of interaction to handle possible conflicts, or reduce redundancies. Examples of these schemes are plan merging [75; 3; 4; 1] and tools to detect the possibility of opportunistic cooperation [5; 1]. The GPGP/TAEMS [45; 24] approach is also an interesting one: in TAEMS, multi-robot plans are represented as network of tasks, each task being assigned to one single robot. TAEMS plans around a notion of influence: how an action of a given robot impacts the actions of another robot. For instance, GPGP/TAEMS is able to reason on the influence of the perception of an information on the path planning process of another robot. The plan is built cooperatively using a blackboard system in which each robot can change the multi-robot plan.

1.2.2 Executing multi-robot plans

The main specificity of multi-robot plan execution is that communication is not be available at all times. This is something seldom taken into account in most systems: for instance, the developers of GPGP explicitly avoided that issue.

Based on their work on teamwork, Tambe et al. developed the Machinetta proxies [61] as a way to decentralize the execution of the team plan. Each system has a representation, through Machinetta proxies of an *estimation* of the involvements the other teammates have into the team plan. If the proxy estimates that a teammate can not be relied upon, the system will act *as if* the teammate actually confirmed it has left the team. This approach is interesting since it uses as much communication as possible, making the estimation as good as possible.

Another approach to handle communication loss is the decentralized classes of MDP planning: DEC-(PO)MDP are centralized planning processes which generates *decentralized* strategies. This means that, once the strategy is generated the robots can act without explicit communication *at all*. This is not very practical at the moment – since most of the time, if communication is possible, the multi-robot system will behave much better by using it. However, the modelling of communication acts in decentralized MDPs is a topic currently under development.

1.2.3 Conclusion

A common architectural approach to integrate multi-robot paradigms is to separate the architecture into three components [31]:

- the *description* of the problem, and the description of the resulting joint plan. The first part is for instance a description of robot capabilities and of the task requirements with respect to these capabilities. The second part is the notion of allocated task or role.
- the *negotiation protocol* through which we build the joint plan: contract net, teamwork, ...
- the *execution engine* which handles the problem of multi-robot plan execution.

Our focus, in this thesis is the *representation and execution* of the joint plans. Since one of our goals is also simultaneous plan execution and modification, we also provide a generic blackboard-like tool supporting plan-based negotiation. Nonetheless, this tools does *not* constitute a negotiation protocol: it constitutes instead a basis for the development of negociation protocols.

1.3 Overall approach

Based on the reflexion we just outlined, we decided to base our approach on a *plan management* component. This component offers a representation of a plan which includes all the robot activities, an execution scheme based on that information, and the tools needed to modify that plan while it is being executed. This plan management component aims at providing a basis for the integration of existing decision-making tools and the development of new ones based on its capabilities: planners, cooperation protocols, ...

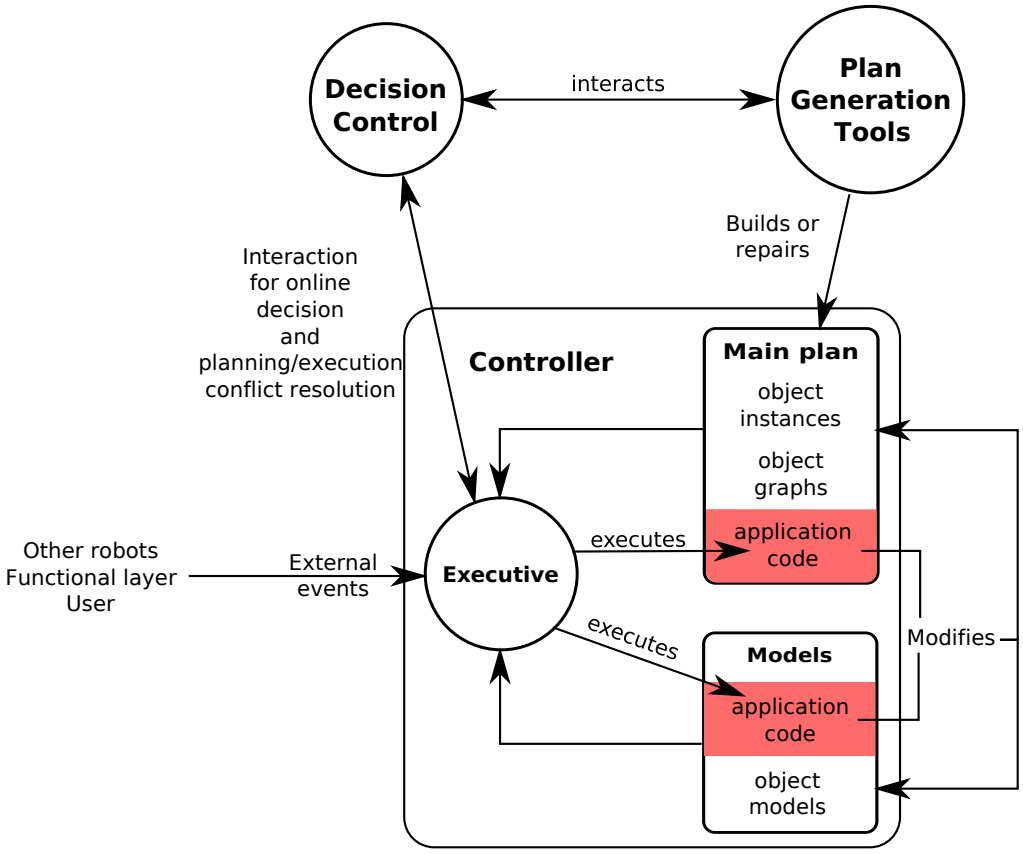


Figure 1.2: The different software components and the data which make a Roby application

Therefore, our plan manager aims at addressing the following issues:

1. represent all the robot activities and the interactions between these activities: avoid the partition of information brought by the separation into layers. Moreover, represent one robot activity in the context of the other robots activities, including the notion of team.
2. provide a generic execution scheme for that plan, in both mono and multi-robot.
3. dynamic plans: allow plan modifications while they are executed, and do so in a multi-robot context. Integrate in that context various plan generation tools, including the plan repair capabilities that these tools can have.
4. reflexion on the need for online decision making: where and how it is needed to make explicit decisions during the plan execution.

In our system, a *controller* – which we also call a *plan manager* – is a complete application tailored for the control of a specific robot. Such an application is made of *models* – the formal description of the objects in the system – and *code* – either generic code written for all controllers or specific code written for the robot.

In a controller, the following components can be singled-out (Fig. 1.2). Each component is directly related to the goals outlined above:

1. the *models* are only data. It includes application code provided by the user, and the *definition* of the different objects the system will be able to manipulate in the plan. The models are then used as a basis for the definition of the *main plan*: networks of objects which are built according to the models. The design of this plan model has been driven by the goals outlined above: multi-robot plans, translation from multiple plan generation tools, representation of all the robot activities.
2. the *executive* is a software component which reads external events and reacts to them, basing itself on the data in the *main plan*. In particular, it calls application code when needed. This executive handles both single-robot and multi-robot execution of plans.
3. the *plan generation tools* are external tools which are called to build new plans asynchronously, or – if they have that capability – to repair their current view of the plan, the repair being then integrated back into the main plan. To integrate these tools, the system provides generic mechanisms to modify the plan safely while it is being transformed.
4. finally, the *decision control* is a software component which is called when the plan execution requires decision: arbitration between conflicting parts of the plan, different possibilities of repair, ... We will see in each chapter that it is involved in all online decision-making, making the decisions whenever it is needed during execution.

To demonstrate the mechanisms presented in this thesis, we have developed the **Roby** software library and the associated application development framework. In this implementation, the controller is a whole application: it is a set of models, planners, decision control, interaction and team-management functions which are used to control a given robot. This controller is written in the Ruby programming language, and we distinguish the *framework* code – the code in the Roby library and framework itself – from the *application* code which is written outside the Roby library, designed for specific needs.

The next chapter deals with the definition of the models and the definition of the plan. Then, chapter 3 describes the executive: what is the process from which we transform the plan into orders for the functional layer. Chapter 4 then describes the plan adaptation capabilities of our system. Finally, chapter 5 outlines some key points of the implementation and some experimental results. Since the decision control component is involved in all the steps of the plan management, its functions are described when they are needed.

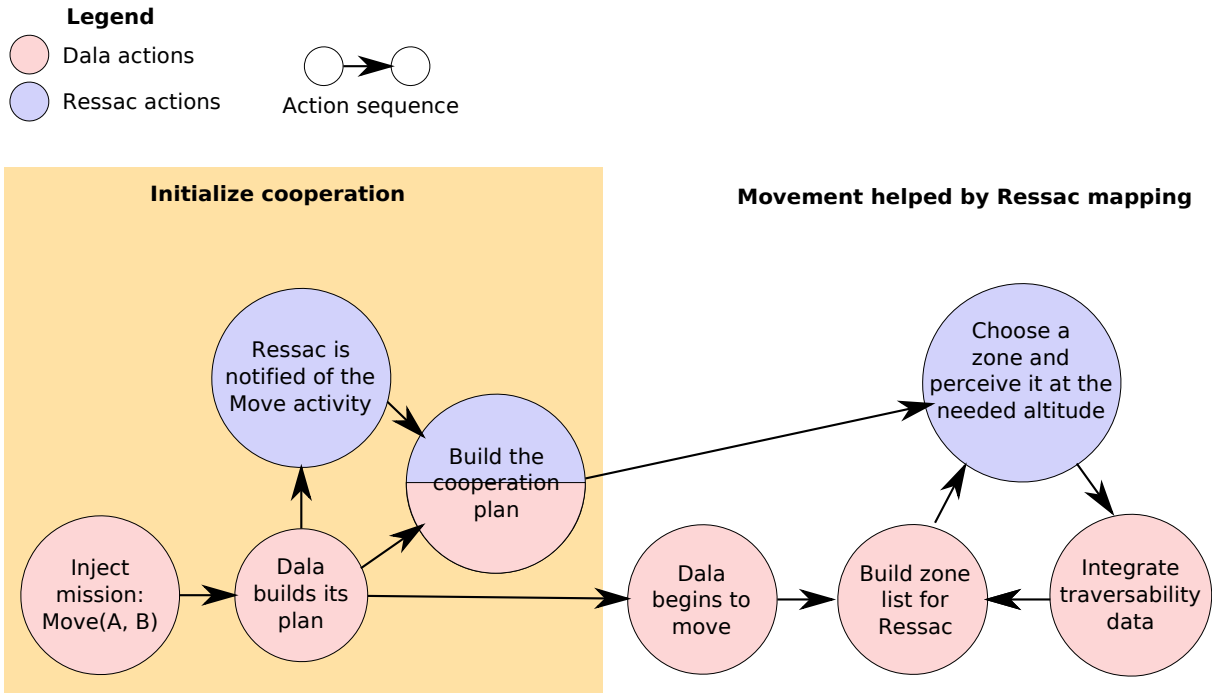


Figure 1.3: The three phases of the rover/UAV cooperation. "Dala" is the name of our rover and "Ressac" the name of the UAV.

1.4 Supporting scenario

To demonstrate the viability of our system, we have designed an experiment of Rover/UAV cooperation. This scenario satisfies the following requirements:

- the functional layer of the rover is complex enough to demonstrate the flexibility of our plan manager.
- the rover/UAV interaction is rich enough to demonstrate the viability of our system to manage multi-robot systems.

This scenario has been extensively tested, both using our simulation system and in the field. The results, as well as key points of the implementation of our plan manager are presented in chapter 5. This section will present that scenario, as well as the capabilities and overall software architecture of the two robots involved in it.

As you will notice, the actual multi-robot scenario is a two-robot one. Implementing the multi-robot communication layer which would have been required by a full multi-robot system has not been possible during this thesis. However, we believe that the mechanisms presented in this dissertation are not limited to two-robot systems and will scale well in multi-robot systems.

1.4.1 Scenario: rover navigation in unknown environment

The goal of this scenario is to make a rover reach a goal efficiently in a completely unknown environment. The Dala rover is able to locally update a traversability map of its environment and plan a path into this map. The Ressac UAV helps the rover by also building a traversability map.

The Ressac UAV is a RMAX helicopter which is owned and operated by the ONERA/CERT lab in Toulouse, in cooperation with which we put this experiment into place. The Dala rover is an ATRV, is owned by the LAAS/CNRS.

On the UAV side, we make the assumption that its movement is free of obstacles – no object detection and path planning is therefore needed for the UAV. Moreover, the quality of the information returned by the UAV is affected by its altitude. To avoid the well-known issue of path planning constantly "switching" between completely different paths, the perception is done at two altitudes: Ressac performs a high-level, low-confidence mapping of the regions for which there is no information at all, thus giving a lot of low-confidence information which is enough to stabilize the path planning algorithm. Once there is information for all the current rover's planned path, the UAV performs a more fine-grained mapping of regions of interest for the rover.

The scenario general timeline is as follows:

1. an operator specifies a global goal in the rover's plan. The rover generates a plan for this goal and starts executing it.
2. once the rover and the UAV are able to communicate, the UAV "notices" the rover's goal, and it notices that this movement depends on traversability information. It proposes the rover to cooperate. Both robots then cooperatively build their joint plan, a plan in which the UAV generates a traversability map for the rover.
3. the two robots execute their joint plan, and react to errors which may appear in each robot's plan and in the joint plan.

The remaining of this section presents the functional layer of Dala since we rely on it for most of our example. Ressac's control, which is presented next, is in comparison much more simple: the plan management system simply sends "zone mapping" orders to a "black box" which handles the UAV movement and the map building process.

1.4.2 Dala functional layer

The Dala rover is an ATRV from iRobot, which is not capable of running in difficult terrain, but can still go into "rough" terrain. We describe briefly three parts of the functional layer presented on Fig. 1.4: the path planning part and the two navigation modalities (for flat and rough terrains). The functional layer uses GenoM, which is a generic framework for integration of modular functional layers [30].

The traversability mapping and path planning processes are as follows:

- from stereovision, a digital terrain map is produced by the Dtm module. This terrain map is a local one: the module exports only a zone roughly 10 meters wide around the robot.
- from this terrain, a virtual robot is placed at each point of the map. From its resulting attitude is deduced a difficulty value in [0 1] [56].
- this local difficulty map is then integrated in a global one by the Bitmap module.
- finally, the Nav module plans a path in this traversability map using a D* algorithm [33].

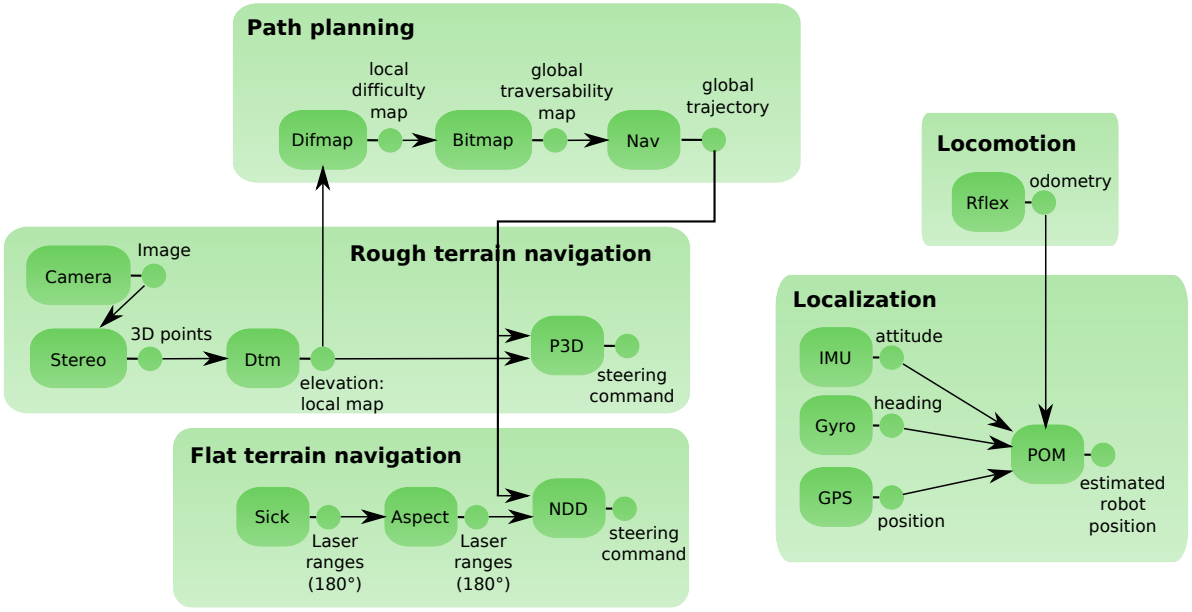


Figure 1.4: Dala's functional layer. The rectangles are the functional modules and are attached to the data they produce. The arrows represent the data flow. The links between the position poster and the rest of the system are not represented. Neither are the links between the motion command and the locomotion module. Note that only one of the two navigation modality is active at any given time.

The rover has two motion modalities. The first one is based on the detection of obstacles by a laser range finder and as such is only available on flat terrain – where there is no problematic obstacles under the laser plane. The NDD local obstacle avoidance method is an evolution of the Nearness Diagram method [48]. The second motion modality is based on the same principle than the difficulty map generation: a local path is planned into the terrain map based on the estimated attitudes of the robot, hence the name P3D [17].

Both modalities are *local*: they do only a short-range planning of their future path. They are used to execute the long range path from the Nav path planning module: this module generates a set of local goals which are executed in sequence by the motion modality modules. Of course, the two modalities are not used at the same time. We will see that our system has the ability to switch between the two motion modalities seamlessly.

1.4.3 Ressac functional layer

The structure of the UAV functional layer is presented on Fig. 1.5. This functional layer is partly based on the layer used during the Ressac experiment described in [28]. From the supervision system point of view, the functional layer can execute very few orders: a movement towards a GPS point or the mapping of a zone, this zone being defined by an altitude and two points.

The main particularity of this layer is that the tasks cannot be interrupted. It has actually be an important point during integration as, once a mapping is started, the supervision system cannot stop it to start another.

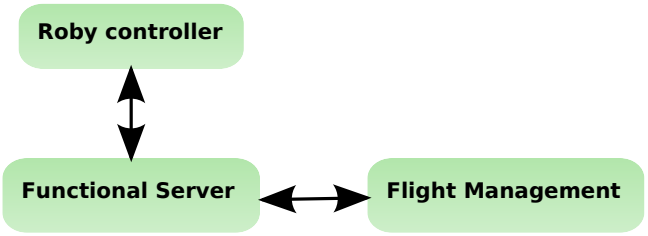


Figure 1.5: Ressac: component architecture. The functional server is the blackbox the Roby controller is interfaced with. The actual UAV control is achieved by a separate fligh management CPU.

2
A Plan Model

As we saw in the previous chapter, being able to represent all the execution context and all of the current robot plans into one single software component helps the integration of multi-mission robots through the use of multiple planners, re-use of supervision code, and promotes the integration of global plan analysis tools not tied to a particular planner.

In this chapter, we describe the plan model used by our software component. The goal is not to provide a generic planning framework, but a model generic enough to represent what's needed for execution, situation representation, supervision and plan adaptation. It has been designed to be simple enough to allow the translation of the execution-related information from other plan models while expressive enough to allow complex supervision schemes.

To design this plan model, we have to define what is a *plan* from a supervision point of view. For the goal outlined above, it should include the following:

- a representation of the situation. This means that not only it must have a notion of *what* the robot is doing (the set of running activities), but also *why* (dependencies between activities), and *how* it got there (history).
- a representation of what may occur in the future. It is usually described as the set of state changes that may happen considering the current situation and/or as a set of events which can be observed next. Note that the two notions overlap: the state changes are often deduced from events (state transitions based on the success of a given activity for instance).
- a representation of how the system will have to react when a set of events occur, or in an equivalent way when a given state is reached.
- since we want to express multi-robot systems, a representation of who is doing what (role).

First, we will describe the objects we manipulate, using our rover as an illustration of the

various parts of the model. We introduce the notions of events and tasks (section 2.1), and define the various relations between tasks (section 2.2). Then, in section 2.4, we show that it is possible to translate various plan models into ours: execution policies, the Hierarchical Task Network (HTN) model [38] and the plan model of the IxTeT temporal planner [43].

2.1 Plan Objects

Our plans are made of graphs of two kinds of objects, events and tasks. The event graph represents the planned *execution flow*: how the system should evolves during execution. The task graph represents the state of the system activities, and how they interact with each other: how they interact now, and how we plan that they interact later. We will not talk here about how the plan is managed during execution (this is presented in chapters 3 and 4): we will only show how events, tasks and their graphs allow to model expressive plans.

Separation of tasks and events, as we do here, is not done in all plan representations: most use time operators between tasks (for instance, running activities in sequence or in parallel). Some representations, like in the IxTeT plan model, do use both tasks and events, but in their case the task structure is not very expressive: most of the information needed to interpret the plan is contained in the event structure[1]. The separation of both structures allows to have an expressive temporal representation (like combinations of events, representation of time constraints, progressive tasks), while having a representation of the relationships between parallel activities (something that IxTeT is lacking for instance). Moreover, since our goal is to provide a plan manager for *multi-robot systems*, parts of the model are multi-robot specific.

We first describe how the execution flow is represented through events and the event graphs, and then we describe how tasks represent the robot activities. Task graphs (also called task relations) are described in the next section.

2.1.1 Representing the execution flow: events

Our execution model is event based: the system execution is represented by a succession of events which represent specific achievements (see Fig. 2.1 for notations related to events). When this particular situation is met, we say that the corresponding event *occurred* or that it has been *emitted*. The system can then be controlled by the set of events which have an *event command*. This command represents the mean to achieve events in a deterministic way: if the command is called, then the event *will* be emitted (deterministic occurrence) in the future. Of course, there can be a delay between the command call and the event occurrence. We can for instance have a e_{brakes_on} event: its command sets the robot brakes and it is emitted when the brakes are actually set on the robot.

As it is commonly done in event-based representations, we distinguish between controllable events and contingent events. In our system, contingent events have no command and the system can therefore not force their achievement: they will be emitted because of situations not controlled by the robot. They can for instance be used to represent non-controllable state

[1] what we call an *event* is called a *timepoint* in the IxTeT plan model

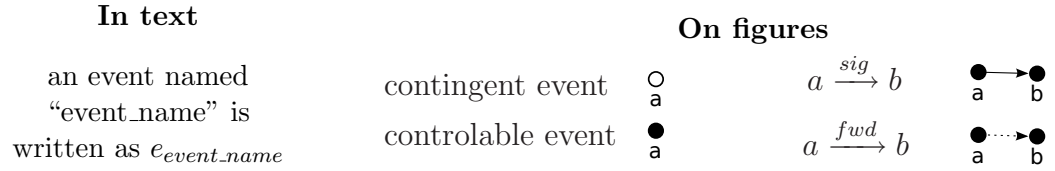


Figure 2.1: Notations related to events

changes like a $e_{low\_battery}$ event. Note also that a controllable event can be emitted because of external conditions: e_{brakes_on} is emitted when the robot's bumpers touch an obstacle.

We can now represent through events what happens during the plan execution. What we are lacking is a way to represent what to do when a particular situation is reached. In our plan model, this is represented by *event graphs*, also known as event relations. There are two event graphs. The first one is the signalling graph: a $e_a \xrightarrow{sig} e_b$ relation in the plan means that the command of e_b has to be called when e_a is emitted (therefore, e_b has to be controllable). This graph therefore represents the *reaction* to events: it makes the robot do something (represented by e_b's command) when another event is emitted.

The second relation is the forwarding graph, which represents generalization between events. A simple example is the end of an activity: let's assume we have a simple activity which can end either successfully or with failure, something represented by two events $e_{success}$ and e_{failed}. We also may simply want to know that the activity did finish, and for that we define e_{stop}. Now, we see that $e_{success}$ and e_{failed} are subcases of e_{stop} : when the activity finished successfully or with failure, it has finished. e_{stop} should therefore be emitted whenever the two other events are emitted. Nonetheless, we cannot use signals here since e_{stop} *already occurred* after $e_{success}$ is emitted. The e_{stop} command should not be called, e_{stop} should simply be emitted. The forwarding relation does just that: when a $e_a \xrightarrow{fwd} e_b$ relation exists, e_b is emitted as soon as e_a is. Unlike for signals, e_b can be contingent.

There is no explicit model about when or how events are emitted: we consider that adding this kind of model would constrain too much what we can integrate in our plan database[2]. The only event state that is represented is the *unreachable* state. An event enters this state when the system knows for sure that it is impossible that it will be ever emitted. Reachability is a common problem in execution supervision, and as we will see later it is needed to track errors in plans: it allows to track problems where the system is waiting for an event that will never be emitted.

[2]of course, it is always possible to add it as an extension of the software component

Now, let's summarize the definitions concerning events:

Definition

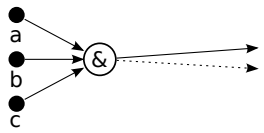
Event definition

- an event is *emitted* or it *occurs* if the situation it represents has been met by the system.
- an event is *controllable* if the system can make sure it will be emitted. In this case, the *event command* is the procedure able to make that happen.
- an event is *contingent* if its achievement is controlled by the environment and not by the system.

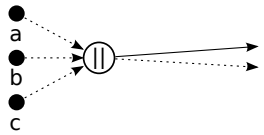
Event relations

- if a *signal* relation $e_1 \xrightarrow{sig} e_2$ exists, then the command of the controllable event e_2 will be called when e_1 is emitted.
- if a *forward* relation $e_1 \xrightarrow{fwd} e_2$ exists, then e_2 is emitted as soon as e_1 is emitted.

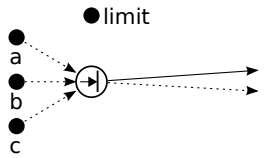
As an example of what can be done using this event model, we will describe how a set of *event aggregates* have been implemented.



The $and(e_a, e_b, \cdots)$ **event** is emitted when all its source events have been emitted. Each source event signals the *and* event. Its command then keeps track of which source event has been emitted and which has not, and emits the *and* event when all sources have been emitted. It becomes unreachable when any of its sources does.



The $or(e_a, e_b, \cdots)$ **event** is emitted once, the first time one of its source events is emitted. This is done by making all source events forward to the *or* event and remove all source relations when the *or* event is emitted. It becomes unreachable when all of its sources are.



The $until(e_{limit}, e_a, e_b, \cdots)$ **event** forwards its sources as long as e_{limit} is not emitted. An event handler, called when the *until* event is emitted, makes the event stop the forwarding by removing all the source relations. It becomes unreachable either when all its sources are or when e_{limit} is emitted.

The $sequence(e_1, e_2, e_3, \dots)$ **event** is a controllable event which calls or waits for the emission of its arguments in sequence. More specifically it does the following: considering an event e_i in the sequence, it either calls the event command if e_i is controllable or waits for its emission if e_i is contingent. When the sequence is called, it handles e_1, and the event sequence is emitted when its last event has been handled. The sequence is unreachable when one of its source events has not occurred and becomes unreachable.

(a) The generic `Roby::Task` task model

(b) The `MoveTo` task model

Figure 2.2: Example of two task models: the generic task model on which all other are based and a less generic one

2.1.2 Representing activities: tasks

Using events, we have a way to express milestones in the plan execution, but we do not represent why these events are emitted: what processes (external or internal) make these event occur. We therefore need a way to represent the system activities. In our plan model, *tasks* are the objects which represent this: a task instance manages the set of events which can be emitted because of the activity it represents. In our implementation, tasks also define event commands, a monitoring routine – which can poll for event achievement – and error handling. In this section, we will distinguish task *models*, which are abstract models of specific tasks, and the task *instances* of a given model, which are the objects actually used in the plan.

A task model is defined by the following:

- a set of events.
- a set of internal relations (signal and forwarding) between its own events.
- a set of arguments, which allow the parametrization of a specific instance based on the task model.

As an example, the generic task model `Roby::Task` represented on Fig. 2.2 has no argument and four events – e_{start}, $e_{success}$, $e_{aborted}$, e_{failed} and e_{stop}. The internal relations express that $e_{aborted}$ is a special case of e_{failed} and that both e_{failed} and $e_{success}$ are special cases of e_{stop}. Moreover, none of these events are controllable (they are contingent), as we cannot define at this level a meaningful command for them. Only one temporal constraint is enforced by the task implementation: e_{start} must be the first event ever emitted by the task, and e_{stop} must be the last. However, these temporal constraints are not directly represented by our plan model. Representing time and temporal constraint is a very important feature for plan representation, but as we stated earlier, we have chosen to design a minimal plan model designed for our needs.

As we just saw, e_{start} is the first event which can be emitted by the task and e_{stop} is the last. Moreover, we say than an event is *terminal* if e_{stop} is reachable from it through the **forwarding** relation: e_{stop} will be emitted as soon as this event is emitted. Non-terminal event are called *intermediate* events and allow, for instance, to represent achievements in progressive processes like incremental mapping, anytime planning, milestones during motions...

A less generic task model is the `MoveTo` task model represented on Fig. 2.2. In the case of our rover, this task model takes three arguments: x, y and yaw. That way, two different motions would be represented using two instances of the same task model with different arguments. On the event side, e_{start} is made controllable since starting the movement is definitely something the model should know how to do. To express the fact that it is possible to stop the task at any time, e_{stop} is made controllable too. From an implementation point of view, the interruption routine is e_{failed}'s command, and the command of e_{stop} calls e_{failed}. $e_{blocked}$, which is not in `Roby::Task`, is emitted when the motion modality does not know how to progress towards the goal.

In our system, some models are abstract models, which means that they define an activity model but not a way to actually make the robot achieve that activity. This is the case, for instance, of the `MoveTo` model we just described. We will see later that specific motion modalities will rely on this abstract model to define non-abstract motion models.

Finally, task instances (or task objects) are defined by a set of task models and an assignation of values to the models arguments. This assignation does not have to be complete: the task instance is *partially instantiated* if its models' arguments are not all set, and *fully instantiated* otherwise. A task instance is then *executable* if its models are not abstract and if it is fully instantiated. e_{start} cannot be emitted on a non-executable task even if the event is itself controllable.

2.1.3 Hierarchies of task models and the substitution principle

The task models build an abstraction hierarchy: all task models but `Roby::Task` have parent model(s). In this inheritance relationship, a task instance of the child model must realize at least the same function than one of its parent. In a plan, an instance of a given task model can therefore be replaced by a task of a child model while not changing the plan results, provided that the two tasks arguments are the same. This is called the substitution principle and has been put into place for the following uses:

- it is possible to represent the fact that two tasks are equivalent from the point of view of the rest of the plan.
- an plan manager does not need to know all specific task models of all the other plan managers. To have multiple robots interact on the basis of their plan, we only need to have a common set of abstract task models.
- it is possible to express abstract plans by including tasks of high-level models and to choose later what specific task implementation to use.
- at the implementation level, this allows to easily reuse code thanks to the inheritance mechanisms of object-oriented languages.

To enforce this principle, we need to express how a child model is constrained by its parent model: in a plan, one must be able to manipulate instances of a child model as if they were of a parent model. We therefore defined the following rules:

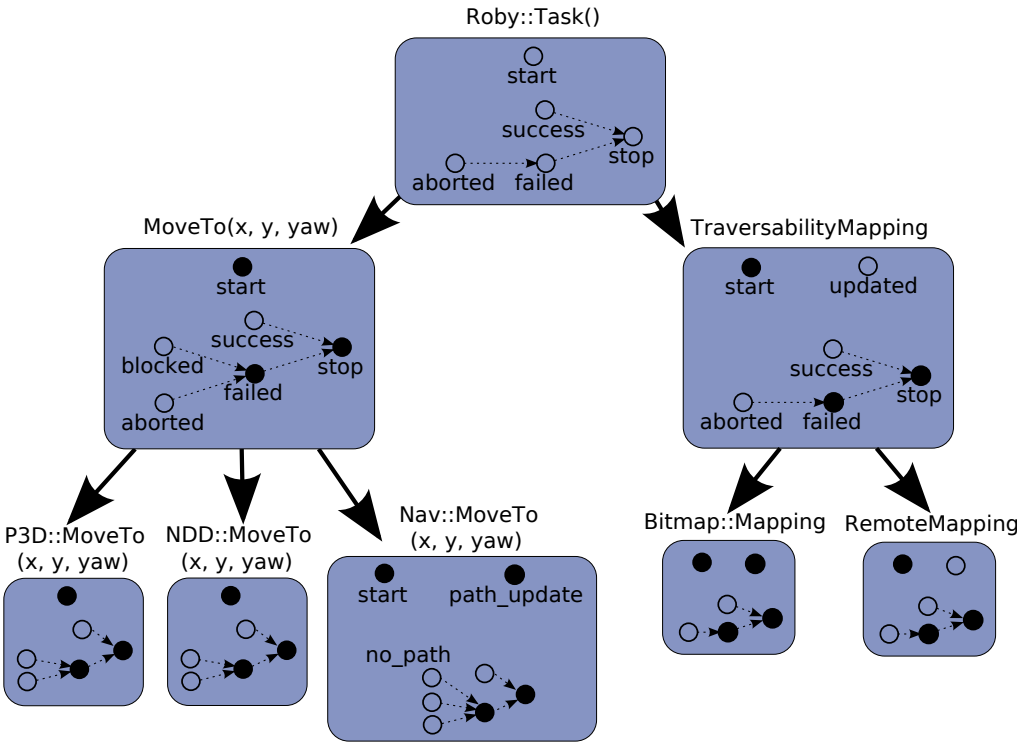


Figure 2.3: Partial view of the task model hierarchy for Dala

- the event set of the child includes the event set of the parent.
- any controllable event of the parent must be controllable on the child.
- the argument set of the child includes the argument set of the parent.

As an example, we can see on Fig. 2.3 part of the task model hierarchy of the Dala rover. In this hierarchy, we can see the three motion modalities Nav, P3D and NDD described in section 1.4. Each of these modalities defines specific `MoveTo` tasks which are children of the generic `MoveTo` task we talked about earlier. The Nav modality differs from the other two because of two new events: e_{path_update} and e_{no_path}. This is because the NDD and P3D motion modalities are reactive modalities while Nav plans a long-term path. The two new events account for this path planning process: e_{path_update} is an intermediate event which is emitted when a new path has been computed, its command starting one iteration of the path planning process, while e_{no_path} is emitted when no path can be found.

On the `TraversabilityMapping` branch, we can see that all mapping activities must define a contingent event $e_{updated}$, this event being emitted when the map has been updated by the algorithm. This non-terminal event represents that the `TraversabilityMapping` tasks are progressive. In the case of Dala's mapping task, `Bitmap::Mapping`, the robot updates its map by doing a perception "here and now" and fuses this perception into the global map. We therefore decided to make $e_{updated}$ controllable, thus modelling the map update as an atomic operation. In the case of our UAV, this is not possible: perception requires some properties of the robot motions, which makes it a more complex action. We decided not to model it as an atomic action by keeping `RemoteMapping`'s $e_{updated}$ contingent.

The same kind of substitution mechanism already exists in other systems. In PRS, for instance, OPs are chosen given the context, and it is common to add arguments to OPs which only distinguish a specific method to achieve some abstract goal. For instance, the multiple motion modality described above could be implemented by defining a MOTION OP whose first argument is the method name (for instance). Since OPs are selected using unification, one could select the right OP given the first called argument. This method, however, does not explicitly define an abstraction hierarchy (there is at most one level of hierarchy possible) and does not allow to reuse code between specializations of the same model.

In task-based representations like HTNs or TDL, the decomposition hierarchy has both the role of refinement (decompose a high-level task into a set of specific actions) and of task selection (choose a specific method for the given task). The plan would not contain directly the information that, for instance, the current motion modality is the P3D one. Instead, one would have to analyze the plan structure to deduce that information. The main problem with this approach is therefore that it gets difficult to distinguish between the two uses, which forbids to share complete plans between agents with only a partial knowledge of the other agents models the way we can in our system.

2.2 Task Relations

While the set of running tasks represent *what* the robot is doing, the task relations will describe *why*. This is not a new notion in supervision systems: in [64], Reid Simmons notes that common supervision systems describe the notion of task hierarchy as a way to express refinement of high level tasks into low-level actions, thus keeping the *why* along with the *what*. This is also present in other plan representations [20; 43] through the use of causal links: the system represents that a goal state is reached by a given set of tasks. However, these systems lack a representation of a hierarchy of abstraction: interpreting causal links as refinement hierarchy is not enough as we will see when we discuss the integration of IxTeT plan models in our system.

What TDL [63] and the Concurrent Reactive Plans system [12] lack is a representation of a task graph: even in mono-robot context, it is very common that one low-level task is being used by more than one parent: for instance, a localization activity is depended upon by many other activities. And it becomes even more important in multi-robot systems: for instance, in our scenario, the motion task of multiple rovers could be linked to the mapping task of the same UAV. Task trees are therefore not enough to represent the activity relations of a whole robotic system, and we choose to manage the tasks in graphs, or more specifically directed acyclic graphs. We will see that using graphs has interesting consequences for plan management. Note that this notion of task graph is necessary to fully represent plan models based on causal links, in which it is common to see one single task enable multiple tasks.

Moreover, we classify task relations into more than one category: using one single relation to represent activity interactions is not rich enough. Using multiple relations type allows to have a higher-level representation of the *why* we mentioned earlier. This section presents how relation allows to structure the plan and the execution. Then, it presents the four task relations

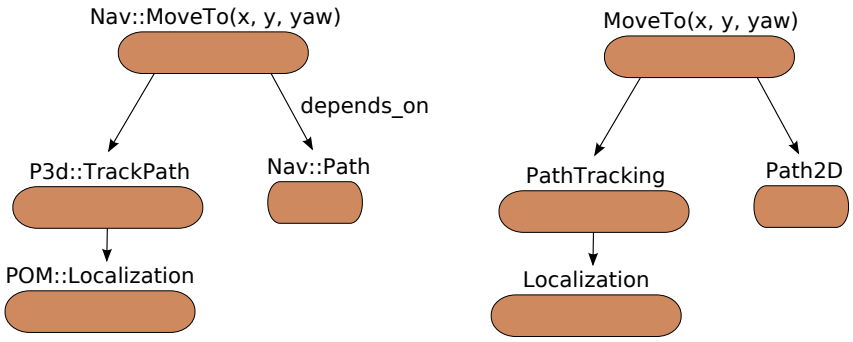


Figure 2.4: **depends_on** relation: **(left)** subtree of a `Nav::MoveTo` task and **(right)** representation of the same tree based on the parent models of the models presented on the left. (Only the part of the plan needed for example purposes is represented here)

defined by the current implementation of our plan manager: the hard dependency and soft dependency relations **depends_on** and **infludenced_by**, the planning relation **planned_by** and the representation of execution support processes **executed_by**.

2.2.1 Defining task relations

In our plan model, we adopted the convention that if a task t_b is the child of a task t_a, then t_b is used in some way by t_a. It means that the result of the t_a task has no direct effect on t_b, while the execution of t_b has a direct effect on t_a, effect which is actually modeled by the relation. From the execution point of view, a $t_a \xrightarrow{rel} t_b$ relation holds two informations:

requirements: what the parent task t_a expects of t_b. Once these requirements are met, the t_a does not depend anymore on t_b.

error conditions: what is an error from the point of view of t_a. This includes failure to meet the requirements, but it is not limited to that: we can also specify undesirable events for instance.

When the requirements cannot be reached anymore, or when an error condition is met, it is represented as an error condition by the software system and we say that the relation *failed*. We will mention what type of error represents these failure to meet relation constraints. The ways to handle these errors are described in the next chapter.

This section presents the four main task relations we defined in our plan model. We will use the subplan required by a `Nav::MoveTo` task to illustrate the usefulness of each relation. Then, we will show how our rover-UAV plan is built using these relations.

2.2.2 Hard dependencies: the depends_on relation

Definition
The **depends_on** relation expresses an action refinement: a task T is **depends_on** a set of child tasks if these tasks are necessary to the achievement of T's function.

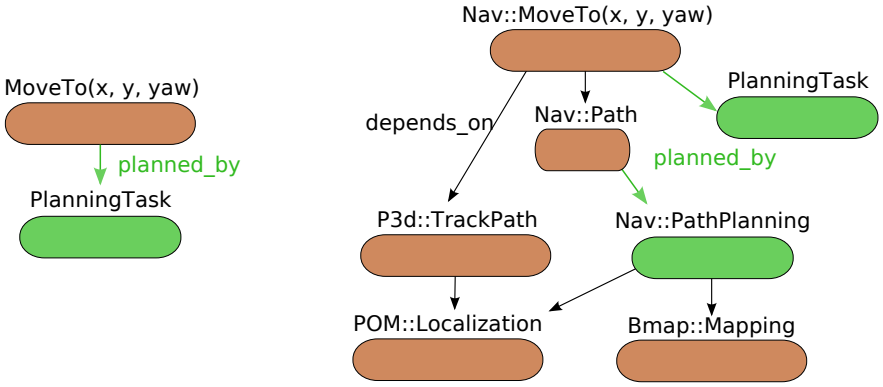


Figure 2.5: **planned_by** relation: representation of an action planning and a path planning activity inside the `Nav::MoveTo` subgraph of Fig. 2.4. The left plan is the initial plan of the rover, with only the rover's mission and the planning task which will generate the plan for this mission.

As an example, let's look at the child tree of a `Nav::MoveTo` task on Fig. 2.4. This motion modality is based on a long-range path planning algorithm (not yet represented on this plan), which updates a `Nav::Path` task. This long-range path is then fed to a local motion modality which is supposed to execute that path. We choose here the P3D motion modality, but the NDD motion modality could have been chosen as well.

A **depends_on** link is defined by the following:

- a task model and a set of arguments defining what kind of task instances can be used as children of this relation. In our example, these are $(\texttt{Path2D}, \emptyset)$ and $(\texttt{PathTracking}, \emptyset)$.
- a set $E_{success}$ of events which enumerate what events the parent task requires its child task to achieve. If all of these events are unreachable, this relation fails. A common value for this set is $\{e_{success}\}$, i.e. the parent task expects its child task to finish successfully.
- a set $E_{failure}$ of events which enumerate what events of the child task the parent does not want to occur: the relation fails as soon as one of these events are achieved. This set is commonly empty: since $e_{success}$ of a task is marked as unreachable as soon as e_{failed} is emitted, we do not need to add e_{failed} to $E_{failure}$.

A failed **depends_on** relation is represented by a `DependencyFailedError` error.

2.2.3 Planning tasks: the planned_by relation

Representing the planning processes is interesting for several reasons: it is possible to represent and handle a failed plan search using the same mechanisms than for the rest of the tasks, to represent how the planning process depends on information-gathering actions, to constrain how much time is allowed before a plan is needed and how the system should react if this constraint is not met,... Moreover, it would be possible to schedule planning processes as some systems do for other tasks, thus taking into account the limited CPU resource and/or the trade-off between plan availability and information quality. Finally, it allows the integration of the planner's repair capabilities by making planning tasks part of the error recovery process.

Definition

A task T is **planned_by** another task P if P's goal is to produces a plan which realizes T function, or adapt T to the changing environment (continuous planning). One planning task can produce an executable plan for many planned tasks while a planned task can have only one planning task.

As an example, let's look at Fig. 2.5. In this plan, we added to planning tasks in the plan on the right of Fig. 2.4:

- `PlanningTask` is the task which generated the plan graph of `Nav::MoveTo`. The original graph, before the plan was generated is represented on the left of the same figure.
- `Nav::PathPlanning` is the task which generates the path represented by `Nav::Path`. This latter task holds the path points as its internal data, and its event $e_{updated\_data}$ is emitted whenever `Nav::PathPlanning` updates the path.

In this plan, the dependency of the path generation task on both the localization service and the traversability mapping is represented. Using a **planned_by** relation here allows to express a different kind of error when there is a planning failure: it is possible, when a planner fails to find a better plan, to keep the old plan and try executing it, even though the execution of the old plan is likely to be suboptimal. There is a trade-off here between aborting the mission and executing a plan which is imperfect but nonetheless executable. This trade-off should be handled by external decision-making tools through the decision control interface.

Therefore, if a planning task fails, a **planned_by** relation will fail only if the planned task is abstract – no executable plan has already been found for it. A failed **planned_by** relation is represented by a `PlanningFailed` error.

2.2.4 Execution agents: the executed_by relation

Modularity is commonly accepted as a must-have for today's modern functional layers: it allows to have a reconfigurable system – including moving towards service oriented architectures [51], to manage the services implementation more easily, and it avoids having bugs in one service impact too much on other services. For all these reasons, we think that it is useful to represent the modules themselves as part of the supervised plan: the modules consume resources, which should be accounted for, and failures must be represented and handled if possible.

Definition

A task T is **executed_by** another task A, called the execution agent, if A is the support process of T. It is also possible for A to be a representation of an underlying hardware module.

In the case of the Dala rover, for instance, each Genom module is represented using a `Genom::RunnerTask` task (Fig. 2.6). This task handles the initialization of the module, allows to kill the module and notifies its unexpected death through e_{failed}.

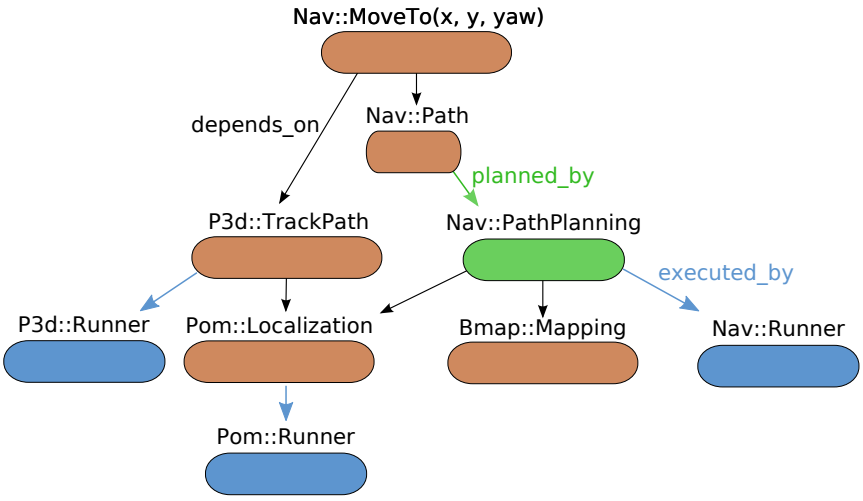


Figure 2.6: **executed_by** relation: introduction of the Genom UNIX processes in the plan representation. Tasks that do not have an execution agent are entirely defined inside the Roby system, while the other represent activities managed inside the Genom modules.

Using the **depends_on** relation here is not possible: the death of a process supporting an activity is not a "normal" failure since we do not know in what state the activity was at the time it failed. In normal operation, when an activity fails, we expect it to have done some kind of cleanup before returning the failure. When the execution agent dies, we cannot expect these operations to have been performed, and this can have important consequences: for instance, the platform can remain in a "moving" state *and the supervision system does not have any means to stop it anymore*.

To represent this difference in behaviour, when an execution agent fails, all executed tasks have $e_{aborted}$ emitted. Specific error handlers can then be defined for this context, to manage the undetermined situations described above.

2.2.5 Soft dependencies: the influenced_by relation

While the **depends_on** relation describes a hard dependency, in which the parent task fails if the child task fails, we also need means to express cases where a task influences one another, but in which a failure of the child task does not mean that the parent task failed. This is a very important feature of the GPGP/TAEMS [26; 23; 45] coordination framework: this framework expresses the effect of a task on a set of characteristics of another task (which are usually a quality metric and the task duration), and then schedules tasks based on that information.

Definition
A task T_1 is **influenced_by** a task T_2 if there is a soft dependency of T_1 on T_2: the results or execution of T_2 improves the result or execution of T_1.

We did not integrate all the decision-related tools built around this relation in TAEMS. However, extending our software system to use TAEMS-based tools (in particular schedulers)

should not be, in our opinion, very hard to do. It would be a very interesting addition when combined with the representation of planning tasks: we could be able to represent the effect of information quality on planner results, and schedule various sensing tasks based on that.

In our example, this **influenced_by** relation is used to express the relation between the rover's mapping task `Bitmap::Mapping` and the UAV's mapping task `RemoteMapping` : if the UAV stops its mapping task – or if its mapping fails for any reason – the rover is able to continue without it, only in a less efficient manner.

2.2.6 Interpreting the task structure: queries and triggers

One of the interesting consequence of specializing task relations as we do is that it is now possible to put semantic on the task structure. For instance, in the rover/UAV interaction we are interested in, one of the functionality of the UAV is to build traversability maps either for itself[3] or for other robots. To handle the latter, the UAV informs the plan managers it is connected to (in our case, the rover's plan manager) that it is interested in `TraversabilityMapping` tasks. Moreover, during a negotiation phase between the rover and the UAV, the UAV could interpret the rover's plan to know how useful the `TraversabilityMapping` task is for the rover. It could then discover, for instance, what the rover movement – which depends on the traversability mapping – is used for.

Our plan manager defines two tools based on this notion of using task structure to represent the semantic of the task relations. The first one is the *query*, which allows to match patterns in plans:

- model and arguments.
- attributes like the executable predicate, or ownership, ... This latter attribute is related to multi-robot and presented later.
- the task state, deduced from what events have already been emitted: running, stopped, finished, failed, ...
- presence of some specific event, and the state of the corresponding event (if it has ever occurred, if it is controllable, ...).
- matching parents and children in given task relations.

For instance, on Fig. 2.6, the `Bmap::Mapping` task would be matched by the following query:

```
Task.which_fullfills(TraversabilityMapping).
    running.
    useful_for(Task.which_fullfills(Nav::MoveTo))
```

The `which_fullfills` predicate being the one matching the model, and the `useful_for` predicate matching only if the considered task is reachable from a task matching the included query (`which_fullfills(Nav::MoveTo)`) through any task graph.

Given a query, the plan manager returns the task set matching it. *Triggers* are built upon this: a plan manager can send queries on remote plans that will *permanently* make the remote

[3] for instance to detect landing areas

plan send matching tasks. This allows a plan manager to track some useful plan patterns on other plans.

2.3 Multi-robot plans

This section presents the multi-robot specific parts of our plan model: what a multi-robot plan is, how each robot activity is managed in a single plan manager and finally how the notion of *role* is represented in our plans.

2.3.1 What are multi-robot plans ?

From our point of view, a multi-robot plan is a form of weak contract between the robots that are interacting through it: a contract, because all the robots did agree on that common plan and should stick to it but a weak one since any robot in the team can decide at any moment that it should leave it. This form of weak contract is something we share with the approach of TAEMS [26; 23; 45]. The other thing we share with TAEMS, or actually GPGP, is the fact that one given robot has only a partial view of other robot plans. This is done for obvious practical reasons: one single robot cannot have a full view on the plan of all other robots, or its plan would become quickly unmanageable[4].

This partial view – which is called the *subjective view* in TAEMS – is built through a subscription mechanism: when robots are building their common plan, they see all the tasks of all robots that are related to that new common plan, and they can choose what tasks they should subscribe to and what tasks they do not need to see. Moreover, a given robot is automatically subscribed to all tasks which are directly related with one of its own.

We now need two things in order to express multi-robot plans: first, we have to express who can change what, and second we have to express the notion of role. The first is needed to handle authority management: since the plan represents what the agents will do in the future, changing the plan is equivalent to send orders to the agents involved in that plan. Therefore, if a plan manager is able to change some parts of the plan, it is actually able to send orders to the robots also involved in it. The second is needed to represent the robots' teamwork through the use of roles: the work of Tambe [68] on teamwork and all the following literature shows that representing roles is an efficient, expressive way to manage teams.

2.3.2 Ownership

In our plan manager, a task *ownership* attribute holds the set of plan managers who are allowed to change the task attributes and relations. From the point of view of a single manager, a *joint task* [68] is a task with many owners, a *remote task* is a task which is not owned by the considered robot and a *local task* is a task whose sole owner is the considered robot. This notion of ownership is already present in the STEAM [61] model of Tambe et. al. Our contribution here is the link with our event-based execution model, presented in the next chapter.

[4] we do not consider here the problem of having two adverse robots giving each other information as limited as possible

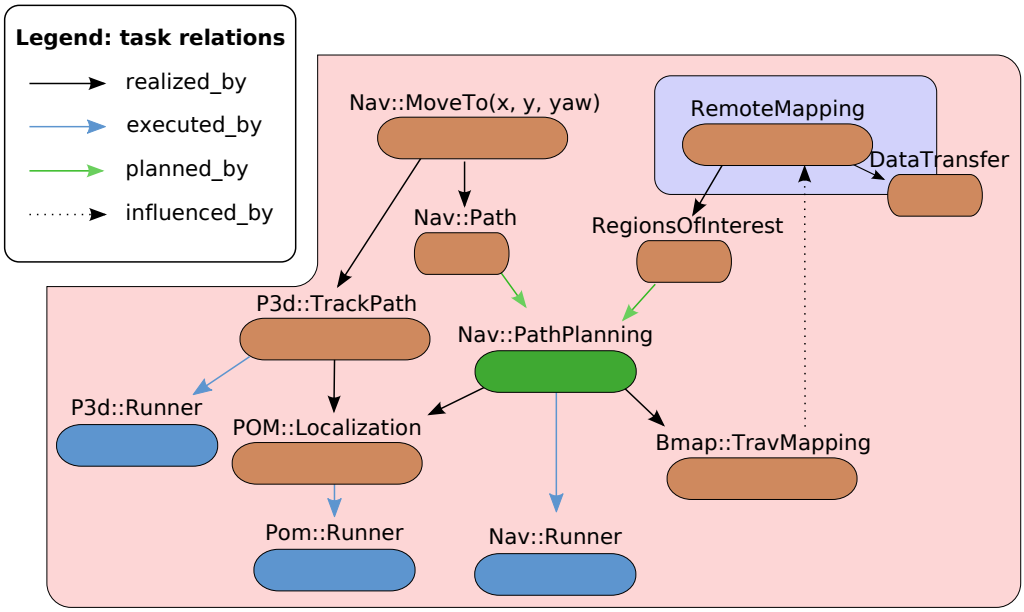


Figure 2.7: Task structure for the rover-UAV interaction from the point of view of the rover. `RemoteMapping` is owned by the UAV and influences the `Mapping` task of the rover. The `RemoteMapping` task is then realized by the UAV's mapping capability and a data transfer task, which is a joint task between the UAV and the rover.

For instance, let's consider the tasks directly representing the rover-UAV interaction on Fig. 2.7:

- a `Bitmap::Mapping` task owned by the rover, which represents the traversability mapping process.
- a `RemoteMapping` task owned by the UAV, which represents the UAV mapping the terrain traversability on behalf of other robots. A `Plid::Mapping` task, also owned by the UAV, will then represent the UAV's traversability mapping itself.
- a `DataTransfer` task owned jointly by the UAV and the rover, which represents the UAV sending the traversability maps to the rover. This task is obviously a joint task of both robots.

2.3.3 Representing roles

To quote Tambe in [70]:

> "A role is an abstract specification of the set of activities an individual or a subteam undertakes in service of the team's overall activity."

One can clearly see that in our plan model, roles can be specified as "an individual or a subteam" set of tasks that are depended-upon the joint task of the team. Representing this notion of role explicitly in our plan model is important because doing so allows to integrate team management tools in our plan manager directly. This section presents the two ways we use to represent roles in our system.

The simple way to represent roles in our plan is to fill a mapping from role names to plan managers, and to express this way what owner holds what role. Note that a robot which has a role must be owner of the task, but an owner can have no role. In the rover-UAV plan of Fig. 2.7, the `DataTransfer` task has for instance two roles: the UAV will have the "sender" role while the rover has the "receiver" role. One drawback of this method is that team management is separated from the plan management: the plan manager cannot replace one task and automatically know the implication of that replacement in the robot teams.

To address that issue, it is possible to represent roles through a combination of the ownership attributes and the task relations. For instance, a subteam B, C of a team A, B, C can be represented by a joint task owned by A, B, C with a child task owned by B, C. The type of the child task then represents what role this subteam is handling in the task jointly owned by A, B, C. This is actually an extension of the approach used by Tambe in STEAM: the role structure is represented by a hierarchy, assigning at each node of the hierarchy a subteam to a role. This notion of role in task models is not present – and is lacking in our opinion – in TAEMS [45]. The COMETS [32; 31] architecture do have roles, but they are not directly represented in the executed plan (what is handled by the Multi-Level Executive in COMETS terms): they are only represented in a separate component which handles interactions with other robots, the Interaction Manager. As already said, we are trying to avoid this kind of separation in our plan manager.

In our system, it is modeled by associating query objects to role names in the model of joint tasks. In the context of the `DataTransfer` task, this is done by adding depended-upon tasks `DataSend` and `DataReceive`, the first one being owned by the rover and the second one by the UAV.

2.4 Translation from other plan models

There are three issues when interfacing with planners:

- express the planning problems in our system (i.e. as tasks).
- translate the resulting plan in our plan model.
- for planners which support it, revision of the current plan when the situation evolves.

This section deals only with the second issue. The first and the third are still an open question in our system. We will first discuss the translation of the IxTeT plan model, for which a prototype has been implemented and a real plan translated and tested. Then, we will discuss the translation of MDP execution policies, with an extension to POMDPs. This translation has not been tested yet because we do not have such a planner and an example policy.

2.4.1 From the IxTeT plan model

The IxTeT planner is a temporal planner whose plan model is related to the STRIPS model. In this plan model, a set of *attributes* evolve over time. A *goal* is expressed as an assignation of a subset of the attributes (i.e. desired values for some attributes). A subset of an IxTeT

plan realizing a multiple goals is represented on Fig. 2.8: the goals are two `PICTURE(...):DONE` goals, and a `COMMUNICATION(...):DONE` one.

In IxTeT, the plan is built using the following objects (see [43] for a full presentation of the IxTeT planner and its model):

timepoint A timepoint is a point in time. All other objects are attached to a timepoint or a range of timepoints to express their place in the timeflow, or a range in this timeflow.

event An event represents a state transition. Events can be controllable or contingent. Example of a contingent event is the visibility window for communication represented by two events on Fig. 2.8(b). Example of a controllable event is the result of actions as the `IDLE -> DONE` event at the end of the `TAKE_PICTURE(...)` action (timepoints `t_1` and `t_2` on 2.8(a)).

hold A hold predicate expresses that the value of an attribute is fixed between two timepoints. For instance the robot should stay put during the communication: there is an hold of the attribute `ROBOT_STATUS` to `STILL` between the timepoints 27 and 28.

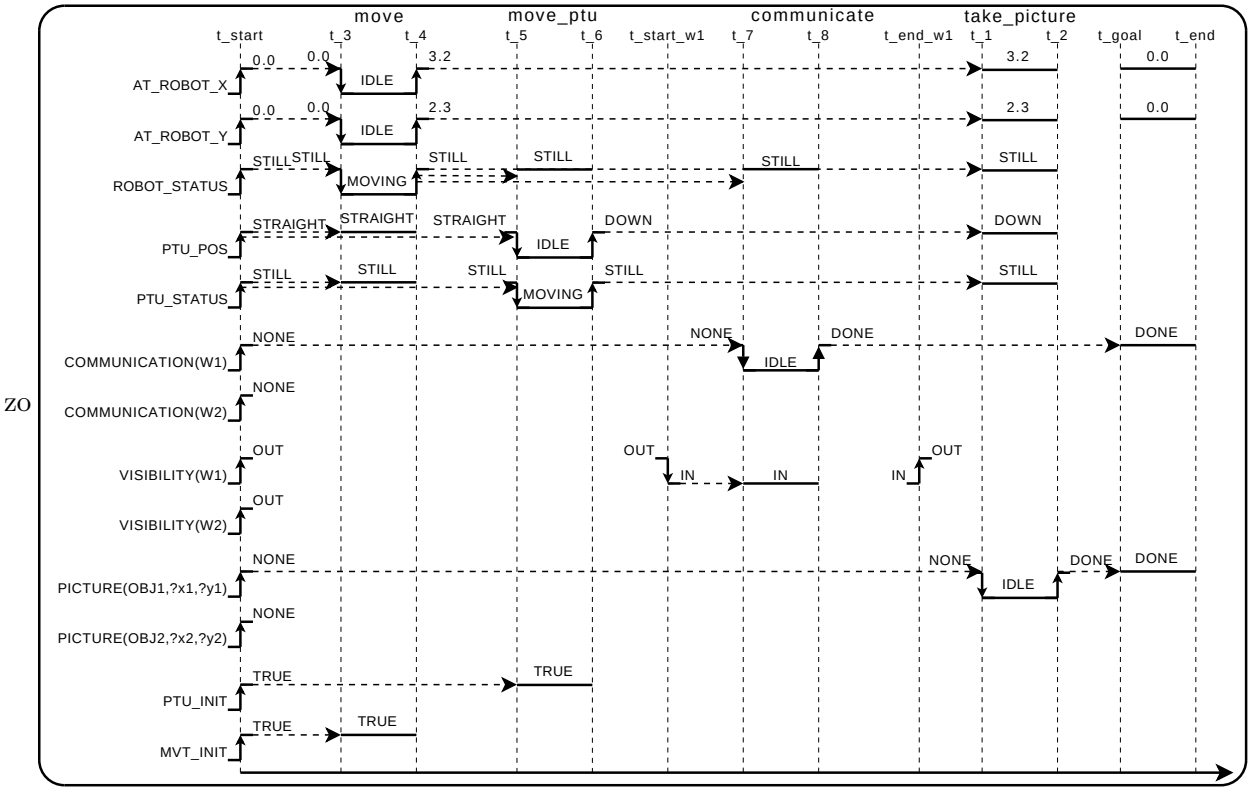
goals Goals are a set of holds: they are described by a state in which the planner should bring the system.

task A task is an action (for instance `TAKE_PICTURE` or `MOVE`). It is defined by a set of arguments: in our plan, `MOVE` is defined by a start point $(x1, y1)$ and a goal point $(x2, y2)$. Task is defined by a name and by arguments which are assigned a value in the final plan. From an attribute point of view, the task is represented by a set of starting events, stopping events and holds. For instance, the `MOVE` task is defined by (column t_3 - t_4 on Fig. 2.8(a))

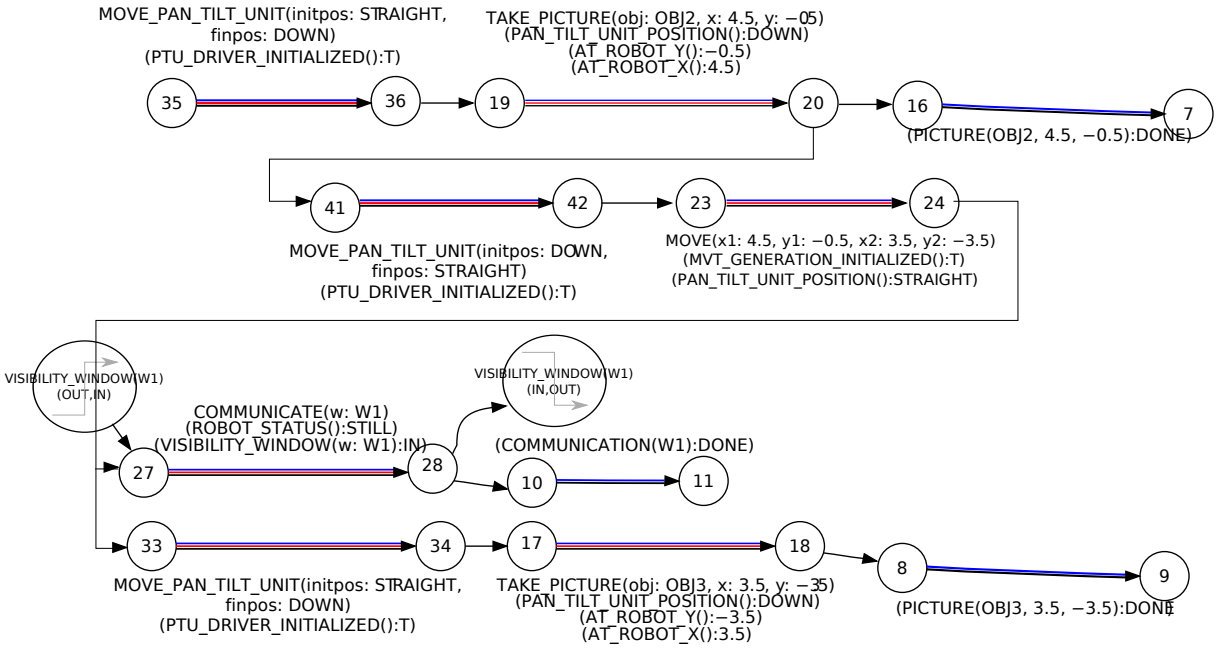
- two starting events which changes the position attributes `AT_ROBOT_X` and `AT_ROBOT_Y` from the current robot position $(x1, y1)$ to `IDLE`, expressing that the movement requires the position to be the one computed during planning, and that this position is unknown to the planner while the robot moves.
- a starting event which changes `ROBOT_STATUS` from `STILL` to `MOVING`. This event forbids to have two movements at the same time.
- a hold which requires that `PTU_POS` to be `STRAIGHT` since the movement requires the cameras to be looking front. For the same reason, there is a hold which sets `PTU_STATUS` to `STILL`.
- a hold which requires `MVT_INIT` to be `TRUE` for obvious reasons.
- end events which sets the robot position attributes to the final position and reinitializes the `ROBOT_STATUS` attribute.

To be valid, all events and holds in the produced plan must be *explained*: there must be another event or hold which sets the attributes to the desired value. This event can be the "origin", i.e. the current state of the robot.

Finally, the IxTeT plan model defines a temporal constraint structure. Since our plan model does not yet integrate such a structure, we will only consider the notion of sequence (i.e. one timepoint which must be executed before another). The future integration of time in our system would allow to explicitely integrate this part of the IxTeT plan model as well.



(a) Chronicles: evolution of the various state variables, representation of events and how events/holds are linked to one another



(b) Ordering of the tasks, goals, holds and contingent events in the resulting plan

Figure 2.8: Example IxTeT plan: a robot is supposed to take two pictures at two different locations. It is also supposed to communicate during a predefined window. It cannot move and communicate at the same time.

To translate these plans, one needs first to define a mapping from the action description of IxTeT into our set of task instances. This can be done either by hand or automatically if some kind of convention is used in the naming of actions and tasks. For instance, translate a `MOVE_PAN_TILT_UNIT()` would be translated into a `MovePanTiltUnit` task (the former is not valid Ruby), while keeping the argument names.

It is possible to derive a notion of dependency from the notion of explanation we described above: a given task T directly depends on the successful execution of all the tasks which establishes the attributes values its events and holds require. This dependency is however *not* the same than the one defined by the **depends_on** relation. In the dependency relation we derive from event explanation, the relation is temporal: the task can only be executed if all the task it depends on have finished successfully. In a **depends_on** relation, the tasks are being executed in parallel. However, it is possible to derive the activity structure as the goals being parent of all tasks which are required for their achievement: a task is generated for each goals in the plan, and the goal tasks depend on all tasks which are needed to reach that goal.

The event structure is built from the sequences extracted from both the structure and the event explanation. Given a task T and its start event e_{start}, we build a and_T event for e^T_{start} in the following manner:

- the $e_{success}$ events of all the tasks T directly depends on in the meaning of IxTeT are added to and_T: T cannot be executed if they have not all finished successfully.
- the e_{stop} events of all the tasks which are just before T in the temporal constraints are added to and_T: T can be executed whenever they have finished, regardless of their result.
- the contingent events which are just before T in the temporal constraint graph are added to and_T as well.

The translation of the plan of Fig. 2.8 is on Fig. 2.9. The combination of task and event relations allows to analyze the global plan structure on failure: for instance, if in the translated plan the first `TakePicture` was to fail – or if the first `Picture` goal was abandoned – the plan would still be executable for the other goals. Moreover, a plan merging mechanism which would detect the redundancy in the new plan (i.e. the consecutive `MovePanTiltUnit` for instance) could remove the redundancies in the new plan.

2.4.2 (PO)MDP execution policies

Generally speaking, execution policies are the most generic interface between planning and execution: it is an expression of the "act" part of the Sense-Plan-Act cycle as a blackbox. Our focus in this section will be (PO)MDP policies, for which the policy returns the action to perform based on either the state of the system or a set of observations (for instance, sensor readings).

A MDP policy gives at each step i the action A_i to perform given the current robot state S_i. Executing a MDP policy is therefore:

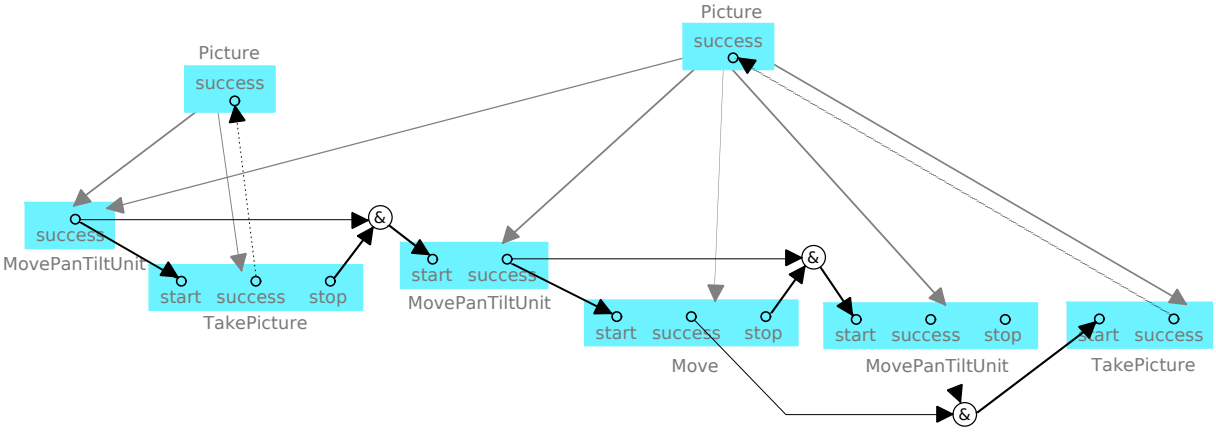


Figure 2.9: Partial view of an IxTeT plan translated into Roby

$$\begin{aligned} \text{update the state} \quad & S_i = ReadState() \\ \text{get the action to perform} \quad & A_i = Policy(S_i) \\ & \text{execute that action} \end{aligned}$$

The simplest way to integrate this decision process in our system is to reactively interact with it: get the current action to perform, and when this action is finished ask the new one to the policy.

A plan-based integration is available when the planner allows to extract the most "interesting" paths of execution. It is not practical to extract all possible paths of execution: policies enumerate all reachable situations or states described by the planning model, which is in general a huge set. However, by making the planner extract for us the "most probable" execution paths we can translate the meaningful part of the policy into our system. This could be implemented by the following process:

- extract the most probable path of execution. This is done by following, at each step, the most probable outcomes of each actions and is completely straigthforward to implement in MDP planners.
- at each step of this central path, extract the branches whose probability to be executed is higher than a certain threshold. Chosing the threshold – and ultimately making it adaptive – is the real challenge here.
- adapt the generated subplan continuously: while the robot executes its plan, it is likely that the subplan extracted by this method will evolve as well.

Using this method, it is possible to represent a partial view of the robot's future actions in our plan manager, and update this view when the situation evolves.

To translate the policy, we assume that we have, as for the IxTeT translation, a mapping from the policy actions into both a task model and a complete assignment to the task arguments,

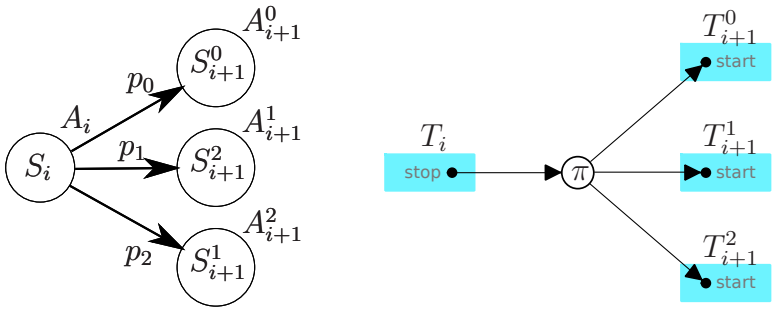


Figure 2.10: Pattern in MDP policy translations: each policy step is represented by a link from the task T_i mapped to the current action T_i and the possible following tasks T^j_{i+1} through a policy event π. During the execution, the event π reads the policy to get the next action which should actually be executed given the measured state of the robot.

thus defining a task instance. It is also possible that the policy actions map to task events, but that will not be discussed here. It is then possible to compute a representation of the policy by using, at each step, the plan pattern represented on Fig. 2.10. This pattern expresses two things: (1) the possible subsequent actions of the robot and (2) that the policy will choose which action to perform during the execution, through the π event.

Finally, during the translation process, a branch which maps to a given system state is reused if that same state is found in another branch (Fig. 2.11): the Markov no-memory property dictates that, if the outcome of the action A_i is the same than the outcome of another action A_j – if $S_{i+1} = S_{j+1}$ – then the policy starting from $i+1$ and the one starting from $j+1$ will be the same.

POMDP policies are more complicated: instead of manipulating a state S_i, they manipulate a belief state B_i: a distribution of probability over the possible states of the system. This belief state being updated by a observation O_i which is a reading of the measurements in the system. The policy execution is then a four-stage process:

$$\begin{aligned} \text{read the observation} \quad & O_i = ReadObservation() \\ \text{update the belief state} \quad & B_i = UpdateBeliefState(O_i) \\ \text{get the action to perform} \quad & A_i = Policy(B_i) \\ \text{execute that action} \quad & \end{aligned}$$

The main issue is therefore the enumeration of the most probable belief states at the end of one action. One this information is known, one can use the same method than for the MDP translation. The most direct way is to enumerate the possible observations – something the POMDP planning model specifies. However, this is only possible if the observation space is discreet and not continuous. At this stage of our reflexion, this is still an open problem.

The representation of (PO)MDP policies we described in this section does not seem very expressive: while the probable outcomes of each actions is represented, there is no information on what will make the policy choose one path or another. However, this integration is a first step

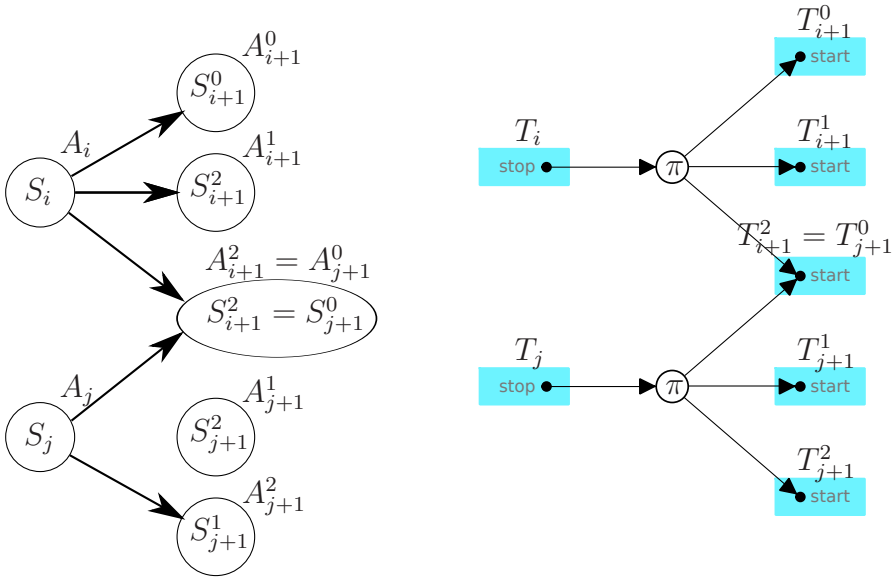


Figure 2.11: Handling of branches during the translation process: in the policy, the outcome of two different branches can be the same state. The following policy would therefore be the same as well.

from which it would be possible to integrate MDP planners – which are not able to handle big problems – in the context of other "classical" planners, other MDP planners and/or robot-robot interaction contexts. Ultimately, it would be possible to export probabilistic information like the probabilities of executing one action or another from the MDP policy, enriching the information in the Roby plan.

This is actually what one could gain by integrating different planners in the same system: use the right model at the right place and implement rich planner/planner interactions through the use of a common plan. It would be for instance possible to implement generic planner/planner interaction schemes: the representation of the possible outcomes of the system can allow to *negotiate* based on that plan: one other planner – or one other robot – could announce that it is interested in some future actions contained in the policy, allowing to update the utility function of the MDP planner, replan, and check if the action is more likely or not to be executed.

2.5 Summary

This chapter presented the way plans are modeled in our plan management system. The main contribution of this plan model is the separation between the representation of activities and of the execution flow: unlike most comparable systems, the execution flow is not defined as an aggregation of tasks (sequences, parallel tasks). The activities are represented through tasks, and their interactions through four main task relations:

- the **depends_on** relation for hard dependencies, and the related **influenced_by** relation for soft dependencies.

- the **planned_by** relation to represent planning processes.
- the **executed_by** relation to represent the external support processes (hardware, ...).

Each task defines a set of events which represent milestones in the task execution, events which can be either *contingent* if they occur only because of external conditions or *controllable* if the system can make them occur.

The representation of the execution flow through events allows a great flexibility in its definition: these events can be aggregated through temporal operators (and, or, until, sequence) to define directly in the plan the appropriate reactions to events. This reaction is defined by the network of event relations:

- the **signal** relation expresses that the command of an event should be called whenever the source event is emitted.
- the **forward** relation expresses the event generalization: its semantics is that the child event (the forwarded event) is a more generic representation of the situation represented by the source event. It is a common tool to represent specific faults in a task: each specific fault is forwarded to the e_{failed} event.

This plan model is also able to represent multi-robot contexts through a combination of an *ownership* attribute and a representation of roles. Roles are represented either directly by mapping the role to a robot involved in the task, or by mapping the role to a pattern in the plan.

The next chapter will present how these relations are used to make the system evolve according to external events and of its plan, how errors are defined and how they are handled, and how all the whole plan execution is managed in multi-robot contexts.

3
Plan Execution

Now that we have seen how one can express a set of activities and an execution flow in our plan model, we explain how plans are executed: how the controller reacts to external events and execution errors. The next chapter will present how plans are simultaneously built and executed.

In the plan manager component, execution is based on a fixed-length execution cycle in which external events are read and propagated in the event structure. When this propagation phase is done, the plan structure is checked for errors (non-nominal situations) and different schemes allow to recover from these errors. Finally, the tasks for which errors remain and the tasks that are not useful anymore are killed. This execution cycle is therefore as presented on Fig. 3.1.

The first section shows how our system propagates events using the two relations presented in the previous chapter (section 3.1). Section 3.2 describes the errors that are recognized by our plan manager, along with the means to handle them in both a proactive and reactive manner. Then, we will see the garbage collection mechanism, which gives flexibility in the management of running activities (section 3.3). Finally, the multi-robot specificities of plan execution is presented in section 3.4.

3.1 Reaction to events

As explained in the previous chapter, the event structure of our plan model describes the execution flow during the plan execution. This section describes how, given a set E of events that have occurred (notifications from external processes, sensor readings, diagnostic), event commands are called and other events are emitted. Plan adaptation, which is also part of this phase of the execution cycle, is outside the scope of our work: our system provides *tools* to change plans as

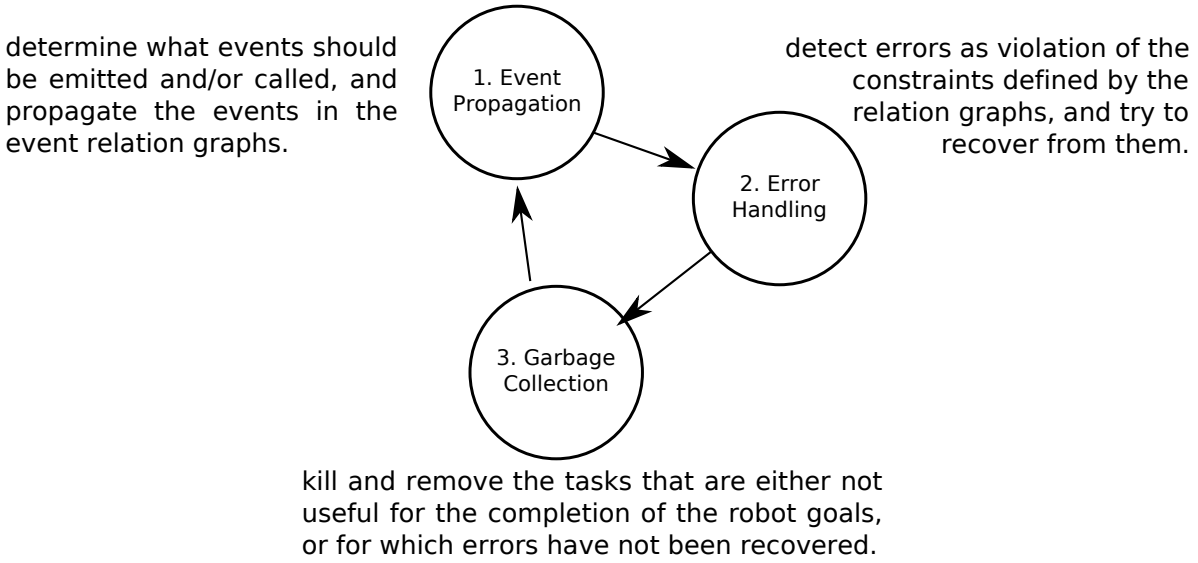


Figure 3.1: Overview of the three main phases of the execution cycle

they are executed, but it does not handle the task of deciding what to change and into what. Interesting reads on this topic includes Beetz [13; 12], GPGP [45], and more generally work on repairing plans [43] and on contingency planning [57].

In our system, event propagation is the process by which, given a set of pending events E, an event is chosen to be called and/or emitted, and how this process continues as long as there are events to propagate. This process is illustrated on Fig. 3.2.

Propagation is in fact not represented by a set of events but by a set of *local propagation steps*, each step representing what operation – call or emission – should be applied on an event, and the set of source events which caused that operation. This section first describes local propagation patterns: how one event can locally affects others during the "local propagation" step on the figure. Then, we present the global algorithm, which, given a set of local propagations, chooses the next one to apply.

3.1.1 Local propagation patterns

Fig. 3.3 shows the graphical representation of what can happen to a single event during the propagation phase. This representation will be used in the rest of this dissertation. Then, when one event is called or emitted, it can affect other events in the following ways:

- the event is parent of other events in the **signal** and **forward** relations.
- the event command calls or emits other events.
- event handlers, which are piece of code that are executed when an event is emitted, call or emit other events.

The event command and event handlers are user code. If an error is detected while executing this code, the error is inserted in a separate error set, this set being processed by the error handling phase.

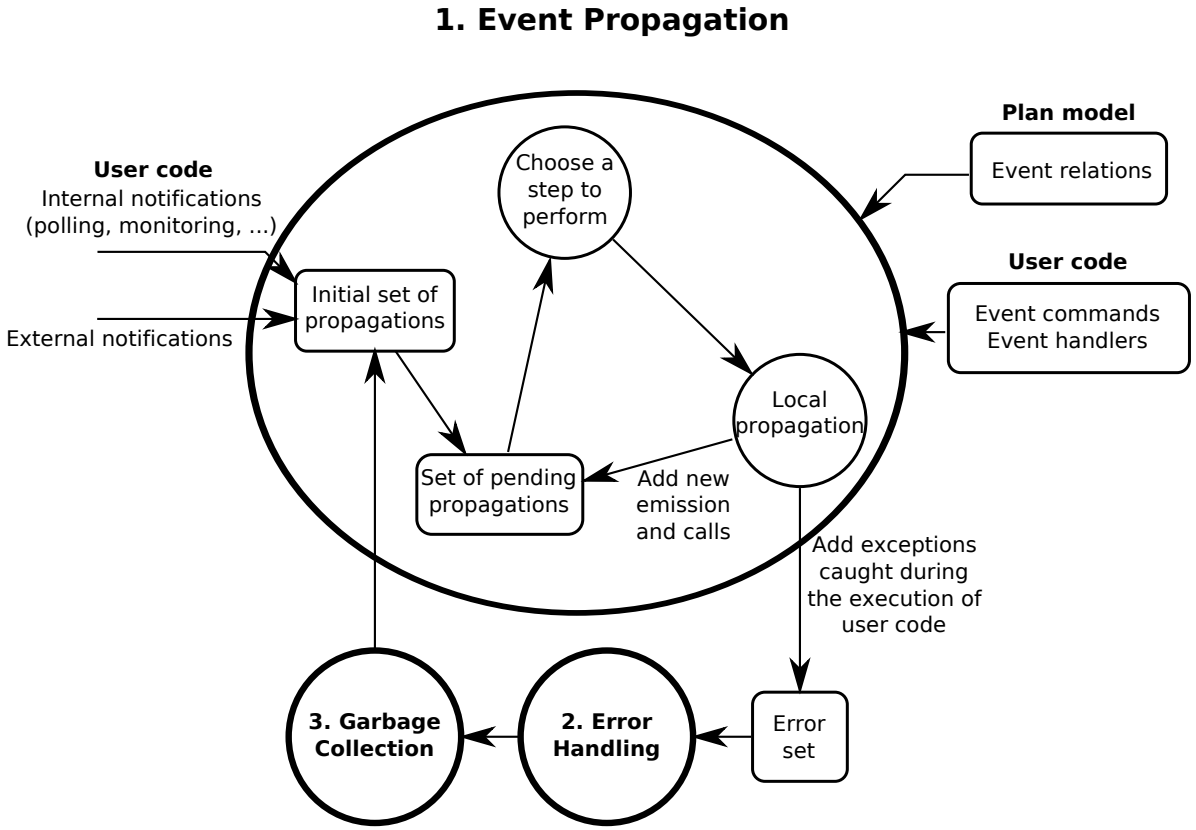


Figure 3.2: Overview of the event propagation phase: events are called or emitted by one by one, based on a set of pending operations, the plan and the user code.

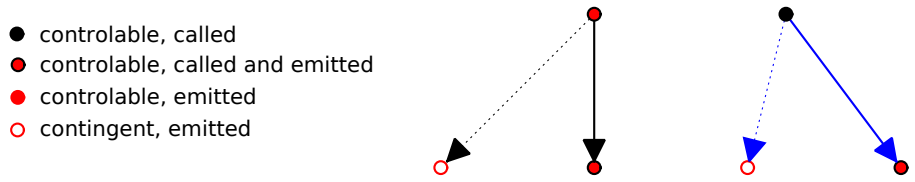


Figure 3.3: Local propagation patterns. Leftmost: the different representation of an event during execution. Left to right: an event forwards and signals another event using the relation graphs or inside the event handlers. Another event emits and calls other events in its command (see that the source event on the right is only called, not emitted)

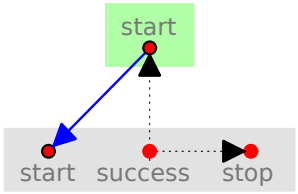


Figure 3.4: Achieving an event by using an external task. This pattern is used to abstract away the fact that the parent task's e_{start} is in fact a complex action.

Fig. 3.3 shows how all these cases are represented by our plan status display. In this representation, a propagation step caused by an event handler is represented in the same way than if a relation was used (for instance, emitting another event from an event handler is equivalent as using a forward relation). Thus, there is actually an explicit event propagation model (the relation graphs) and an implicit one (the effect of event handlers and event commands). This allows to implement more complex propagation schemes than what is allowed by the event relation model (for instance, chosing an event to emit based on the current robot state) without having to add new types of events and having to modify the propagation engine. There is of course a trade-off here between simplicity and the completeness of the explicit model, which is the only one plan analysis tools can access.

Another useful pattern is the achievement of a given event by an external task (Fig. 3.4). This is a variation on calling other events from an event command presented earlier, where the event command of e_{start} calls the start event e_{start} of another task, and emits when the $e_{success}$ event of that same task is emitted. It seems, at first, an inconsistency: a task, which is non-atomic and whose result is non-predictable, is used to perform the action of an event, which is both atomic and predictable. From our point of view, this pattern is a way to represent a hierarchy of abstraction: at a certain level (the parent task), an action is represented as being atomic (its event e_{start}), but at a lower level (the child task), this action is not atomic. As described later, our system is able to handle failures that could be generated by the use of this pattern.

A real-world use of this pattern is, on our rover, the `Pom::Localization` task. The startup (Fig. 3.5) of this task is based on three stages:

1. motion and sensor estimators, which respectively provide separate estimates of the rover and sensors positions, are started. When all estimators are started, $e_{started\_estimators}$ is emitted.
2. `EstimatePosition` produces a current estimate of the rover position. This estimate can be based on GPS readings like it is done here, but also on *a priori* knowledge – like the last robot known position – or other position estimation methods.
3. `Pom::Localization`'s e_{ready} event sets the current position and starts fusing the estimations. This is an example of the pattern described above: e_{ready} is actually achieved by a sequence of two subtasks.

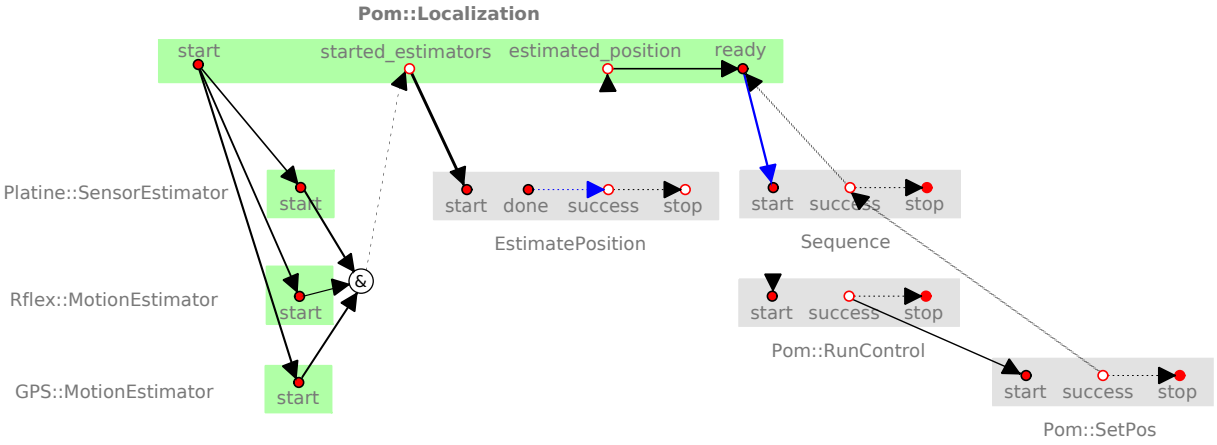
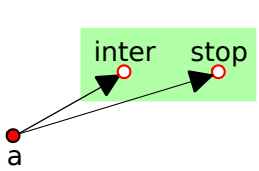
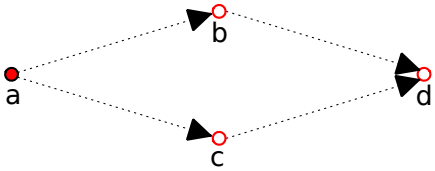


Figure 3.5: Initialization of the `Pom::Localization` task for our Dala rover. This figure shows the three stages of initialization outlined in section 3.1.1. The whole initialization spans more than one execution cycle.



(a) a partial ordering constraint should be added between e_{inter} and e_{stop} so that e_{inter} is called before e_{stop} when e_a occurs



(b) the propagation algorithm must ensure that d is emitted last, or it will be emitted twice even if the situation it represents occurred only once.

Figure 3.6: Two event structures which require a global propagation phase. In both cases, the event propagation must follow a specific order

3.1.2 Global propagation algorithm

During execution, many events can occur in the same cycle and they must all be propagated at the same time in the event graph. The problem with this global event propagation is that it must enforce a partial ordering of events:

- for event commands and handlers to be usable from the programmer point of view, it should be possible to assume some partial ordering of events.
  Let's assume for instance that we want the plan manager to enforce the following rule: "no intermediate event shall be emitted after e_{stop} has been". A natural way to implement that is to check whether e_{stop} has been emitted or not whenever an intermediate event e_{inter} is being called or emitted. This fails on Fig. 3.6(a) if there is no ordering between e_{stop} and e_{inter} : e_{stop} can be emitted first during propagation, and e_{inter}'s call will then be considered as an error.
- in diamond patterns like on Fig. 3.6(b), the event d should be emitted only once since the situation it represents has been reached only once. The use of the *and* aggregator is not a solution: we still want d to be emitted if only one of b or c are emitted.

In our scheme, the propagation is directed by a causality directed acyclic graph (DAG) *PropagationOrdering* in which there is an edge between two events e_1 and e_2 if e_2 can be emitted because of e_1's command or emission, including the event handlers. Obviously, this ordering graph is a superset of both the signalling and forwarding graphs. Based on that model, the event propagation algorithm outlined in Alg. 3.1 marks an event as emitted after all its parents and before all its children.

Algorithm 3.1: Global event propagation which calls and/or emits events in the plan based on an initial set of external events (observations) and the signalling and forwarding graphs.

Input: two initial set of events: E_{calls} is the set of events whose command should be called, and E_{emit} the set of events which should be emitted
function: `topological_sort(`*graph*`)`: returns the topological ordering of *graph*. In this ordering, the first element has no parent and the last element has no child. *graph* must be a DAG.
function: `call_event(`e`)`: call e's command. Updates the event sets if event commands and/or event emission is caused by this call.
function: `fire_event(`e`)`: mark e as emitted and call its handlers. Updates the event sets if new calls and/or emissions are caused by its handlers.

```
while one of E_call and E_emit is not empty do
    Ordering = topological_sort(CausalityGraph)
    e = minimal element of E_emit ∪ E_call in the order defined by Ordering
    if e ∈ E_call then
        call_event(e)
        remove e from E_call
    else
        fire_event(e)
        remove e from E_emit
    end
end
```

Note that it is possible that an event is present in both event sets. In that case, we do call the command and let the emission in E_{emit}. This behaviour ensures that if the command emits the event immediately, the event is emitted only once.

3.1.3 Explicit and implicit model

Our system, like most systems, has two models:

- the first one, the *explicit* model, is the one the system can reason about. In our case, it is the definition of task models, the arguments of task instances and the relation graphs.
- this model is a subset of an *implicit* model. In our case, the implicit model is hidden in the user code our plan manager calls during the execution. Since this user code is for instance allowed to emit events, call commands or test whether or not an event has been already emitted, this part of the model can be important for the whole system behaviour.

Therefore, unlike what exists in synchronous languages, we cannot offer strong guarantees about the system behaviour: parts of the event propagation is by design hidden in commands and handlers. Moreover, the programming language we use is not a formal language and as such cannot be subject to formal program analysis. For instance, in the case of the *until* event a $e_{limit} \rightarrow until(limit)$ is added in the *PropagationOrdering* graph to make sure that if both e_{limit} and a source of the until event are emitted in the same cycle, then e_{limit} will be emitted before the *until* event is. *until* will therefore not forward its arguments in this cycle. We lost the ability to prove the system behaviour for practical concerns, but we could get that ability back by constraining the allowed actions in the user code.

3.2 Error management

Robots are autonomous agents evolving in dynamic environments. Because of the "dynamic" part of the environment, making a perfect (also known as "universal") plan is nearly impossible: plan generation manipulates simplified models of the environment, the environment is only partially known (sensor reading are noisy, state estimation is also based on imperfect models and as such is subject to errors, ...). As a consequence of all this, systems designed to control robots *must* be able to represent and handle failures.

There are mainly three different approaches to this problem, which are not mutually exclusive (Beetz hierarchical controllers do for instance both 1 and 3):

1. in hierarchical controllers [62; 63; 64; 40; 12] an error is either a localized event generated by a task or a constraint which is violated by some task. The *exception propagation* mechanism consists in going up in the controller hierarchy to find an *exception handler*. Exception handlers are piece of application code responsible to find an appropriate response to the problem.
2. the plan itself takes into account multiple path of execution. In classical planning, it is done by conditional or contingent planning [57]. This approach is also inherent in probabilistic planning approaches [60]. Note that this approach does not preclude the use of other error management schemes: the generated plans will *not* handle all possible errors – only the most likely ones.
3. the plan does not take into account the problem, but the planner which generated it is able to either repair the plan, or generate a new plan, whenever the current execution violates constraints described in the plan [43].

In our opinion, the three approaches have specific interests, and our system implements all of them. This section describes first the various errors that are detected in our system, and how they are represented. Then, we present how our system offers these three approaches of error management in the context of our plan model.

3.2.1 Error definitions

An error is defined by two things:

- a *type*, and associated data, which describes the specific error.
- a *point of failure* which is the object in the plan the system determined as being the cause of the error. It also allows to determine where a repair is needed.

In our system, errors can have three origins:

- they can come from the plan management code itself, in which case it is most likely a bug.
- they can come from user code: code in event commands or handlers, or some polling code associated to a task. It can for instance be a bug, or because the code performs itself some internal checks and one of them failed.
- they can come from a constraint violation: during the plan execution, some task violated a constraint which was specified in the plan.

This section presents first how code-related errors are detected and how they may be inserted in the error-handling process. Then, the errors related to the violation of the constraints defined by our task relations are described. For each error, we describe how its point of failure is determined.

3.2.1.1 Code-related errors

When we talk about an error in *code*, we are essentially talking about the exception mechanisms which exist in most modern all-purposes language[1]. The language we use for our implementation, Ruby, has this kind of control construct and when we refer as an error in user *code*, we talk about an exception which has been raised by user code and has been caught at the boundaries between the "plan management code" and the "user code".

Errors in the plan management code itself are in general very difficult to properly handle, as it would mean that we have a model of the consequences of all possible errors on the overall system. There are however places where it is possible to determine that errors in the plan management code have effects only on a single event or task. The *critical* path is then defined as the code for which we cannot determine the effect of the error. If such an error occurs, the only alternative is to terminate the plan manager and assume that some higher level mechanism of fault-tolerance will take over.

In the case of event handlers and event commands, the point of failure is obviously the corresponding event. For these, the following errors are defined:

`CommandFailedError(event, failure)` is generated when the command code of *event* raised the $failure$ exception.

`EmissionFailedError(event)` is generated when the command of *event* has been called, but it is impossible for the event to be emitted.

`UserCodeError(object, failure)` is generated when controller code associated with an event or task raises an error. $failure$ is then the failure object raised by the code. The point of failure for this error is *object*.

[1] see `http://en.wikipedia.org/wiki/Exception_handling` for an introduction

3.2.1.2 Constraint-related errors

For the constraints defined by our task relations, the following errors are defined:

`DependencyFailedError(parent, child, reason)` is generated when a **depends_on** relation is violated. *reason* is either an event from the failure event set of the **depends_on** relation, which has occured, or the set of reasons why all positive events are unreachable: when an event e_u becomes unreachable the system can give an event e_f whose emission is the cause of e_u's unreachability, if such an event exists. In this case, e_f defines the reason and point of failure of the error. Otherwise, the *reason* is task-specific and the point of failure is the task itself.

For instance, let's assume a simple **depends_on** relation $PickUp(object, x, y) \xrightarrow{depends\_on} MoveTo(x, y)$ where `MoveTo`'s $e_{blocked}$ is emitted. In that situation, the **depends_on** relation fails because $MoveTo$'s $e_{success}$ is unreachable and the system blames the most specific error event for it; the most specific event which is emitted and forwarded to e_{failed} (for instance $e_{blocked}$, see Fig. 2.3 on page 39). The error is therefore represented by `DependencyFailedError(`t_1`,` t_2`,` $e_{blocked}$`)`.

`PlanningFailedError(task, planner)` is generated when a **planned_by** relation is violated. The point of failure for that error is the task.

3.2.2 Handling errors

Code-related errors are detected in the first stage of the execution cycle. However, the system does not immediately react to them: they are handled during the second phase of the cycle along with a set of errors detected by global plan analysis procedures which run at the beginning of the second phase of the execution cycle: the error handling phase (Fig. 3.7).

3.2.2.1 Handling errors during the event propagation

Since the error detection through plan analysis is done *after* the event propagation stage, it is possible – if repairing the plan is instantaneous – to handle some errors without using the special tools designed for that. For instance, our rover's operating system is not real time. Therefore, under heavy CPU load, the locomotion control module Rflex sometimes determines that its command source (P3d for instance) did not update the command, in which case it stops the robot and fails with a `POSTER_NOT_UPDATED` message. In our task model, this error is mapped to a corresponding $e_{poster\_not\_updated}$ event and since it is really a benign error, the handler of this event simply starts a new Rflex control task `Rflex::TrackSpeedStart` and replaces the failed one by the current one (Fig. 3.8). Because of this operation, the task failure is not seen by error management.

3.2.2.2 Plan repairs

For each error, the plan is first checked for *plan repairs*. A plan repair is an association between a set of points of failures S_{fail}, a running task T and a timeout t_{max}. As long as the task is running, an exception whose point of failure is included in S_{fail} is ignored. The plan repair is

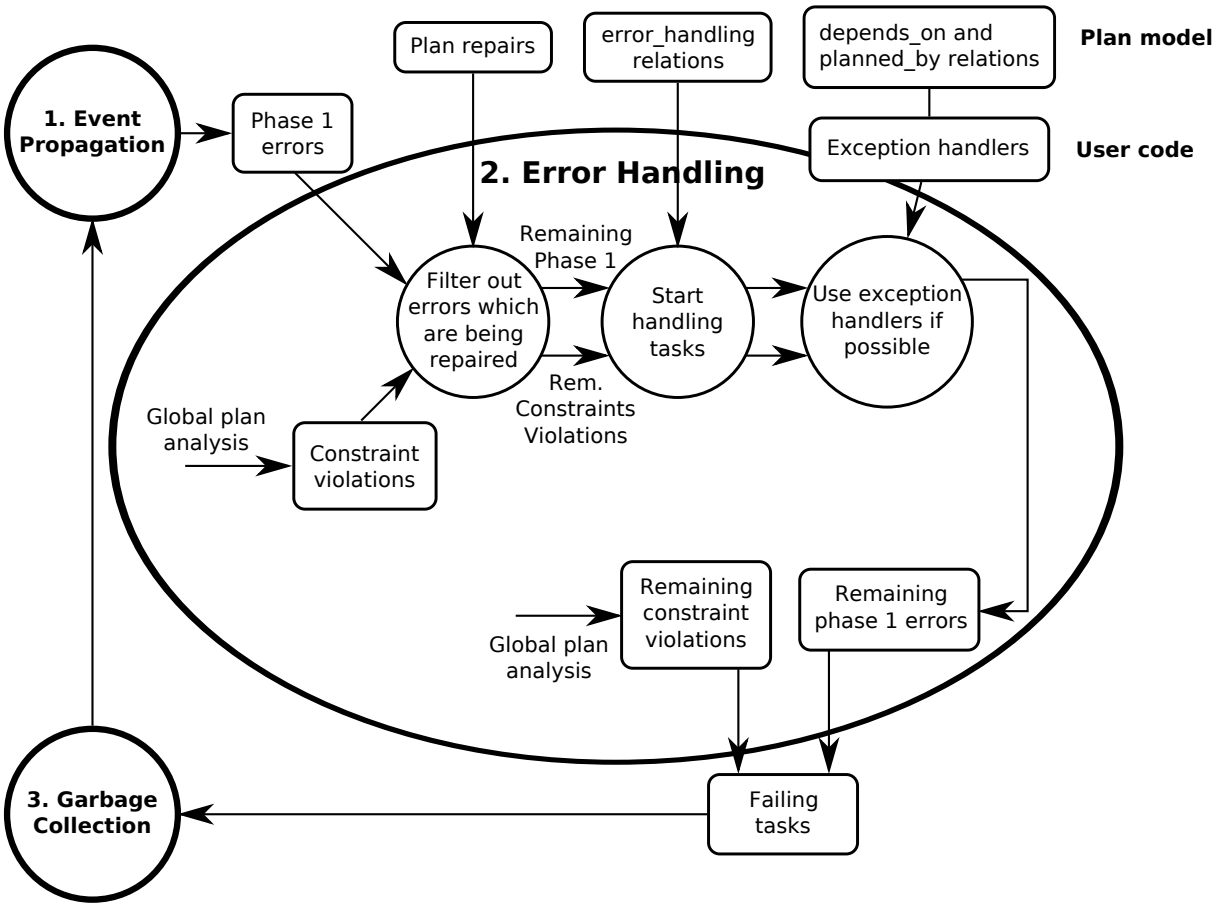


Figure 3.7: Overview of the error handling phase. The errors that are detected are filtered through the three steps of the error handling phase. For the remaining errors, we mark the corresponding tasks for garbage collection

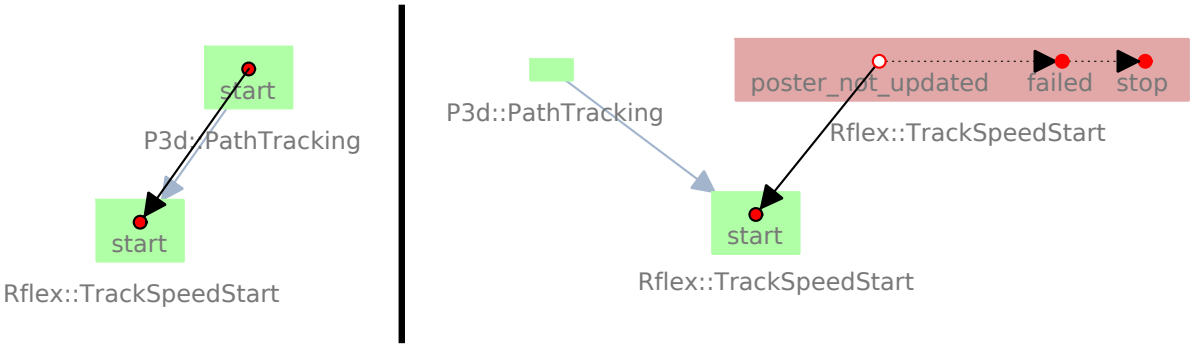


Figure 3.8: Repairing the plan during event propagation: the $e_{poster\_not\_updated}$ event of `Rflex::TrackSpeedStart` is a "spurious" error. To handle it, an event handler is defined on it which simply replaces the failed task instance with a new instance of the same model. **(left)** the initial situation and **(right)** the repaired plan

removed as soon as the task finishes and/or the timeout is reached. Once the plan repair is removed, the corresponding errors are not inhibited anymore.

If no repair exists for the error we consider, the system looks for a **error_handling** relation $t_1 \xrightarrow{error\_handling} t_2$ which is defined by a set of events $e_1, e_2, \ldots$. Such a relation exists if t_2 is able to handle the situation where any event $e_1, e_2, \ldots$ of t_1 is the point of failure of an error. If that situation is encountered, t_2 is started and a plan repair is automatically added.

One specificity here is that the plan repair is associated with all events that are forwarded to the given point of failure: since the forwarding relation has an abstraction semantic between events, we consider that a repair on a more-generic event automatically handles the ones for less-generic events. So, for instance, **error_handling** relation on the e_{failed} event of the `MoveTo` task we talked about earlier would also handle the $e_{blocked}$ event, but a **error_handling** on $e_{blocked}$ does not handle errors in e_{failed}. If more than one **error_handling** relation applies, the decision control component is supposed to choose one for the executive.

As an example, let's look at the handling of the `P3D::TrackPath` task in our controller, in the context of the Nav long-term path planner. $e_{blocked}$ is emitted by P3D when the subgoal given by Nav is not reachable according to P3D's model of the terrain. To handle this situation, we mark the zone in front of the robot as "non-traversable" and regenerate a subplan for the `P3D::TrackPath` instance (Fig. 3.9). However, the issue of *starting* that new subplan is task-specific: the planner and/or the parent task are supposed to have defined event handlers able to deal with that. In our case, we can simply start the new subplan. Our system also provides a plan merging mechanism, which is presented in the next chapter, to help dealing with that issue.

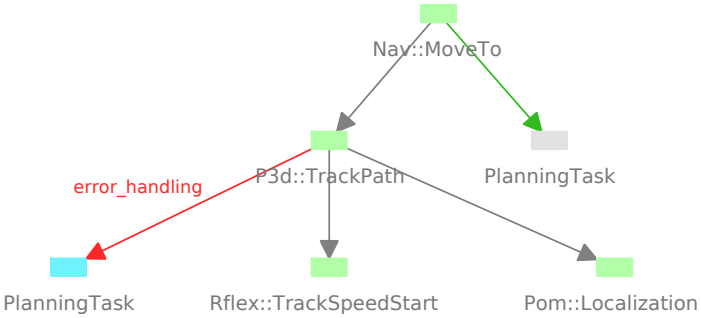
3.2.2.3 Exception propagation

If no plan repair exists and no **error_handling** relation applies, the error is managed by *exception propagation*. In our system, exception handlers are associated with task models. We therefore need to associate all errors with a task:

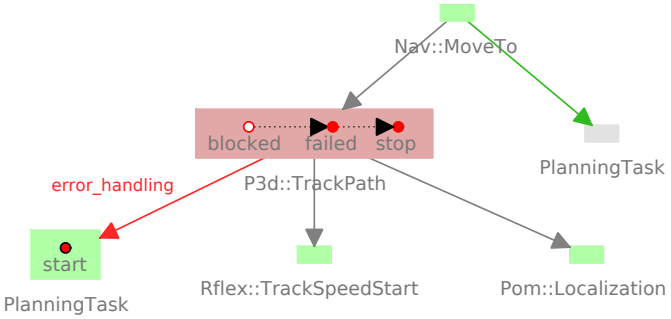
- if the failure point is a task or a task event, the error is associated with that task.
- if the failure point is an event outside a task, the error is associated with all tasks whose events are linked to this event through the event structure.

Once this is done, we propagate each error up in the **depends_on** graph. At each level, we check if the task model of the considered instance defines an exception handler which handles the error. If a handler exists, the propagation is stopped. Otherwise, we look for one on the planning task if there is one, and finally go up in the graph if the planning task does not handle the error either.

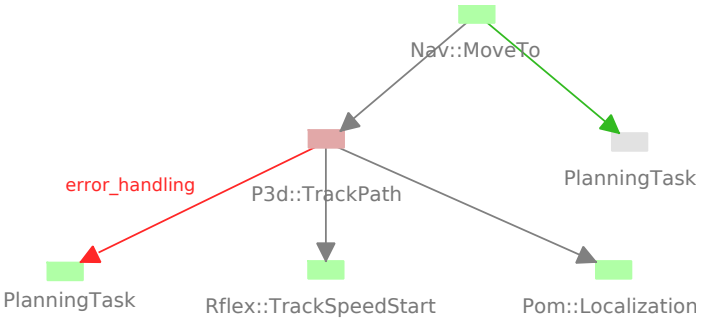
However, unlike other plan models which define exception handling, we do not manipulate trees but graphs. There is therefore the possibility that, during the propagation, two different branches have two different handlers which can handle the error. Such cases are taken into account through Alg. 3.2. The only problematic situation is to decide what to do when more



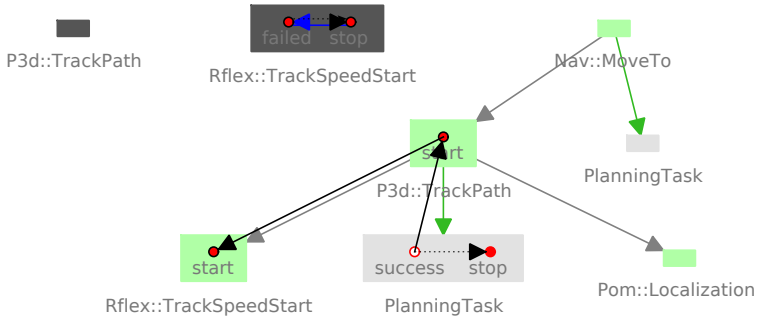
(a) Starting point: the **error_handling** relation in red handles the $e_{blocked}$ event of `P3d::TrackPath`



(b) The `P3d::TrackPath` task emits $e_{blocked}$ to announce that there is no path, according to its model, to the subgoal provided by the Nav path planner. This error is handled by the `PlanningTask` task



(c) While the new `P3d::TrackPath` subplan is being generated, the error is inhibited



(d) Once the new `P3d::TrackPath` subplan is generated, it is inserted and started to replace the one that failed

Figure 3.9: Example of a **error_handling** relation allows to manage the $e_{blocked}$ event of our P3d motion modality

Algorithm 3.2: Propagation of an exceptions in the task graph. The basic idea of this algorithm is to get a propagation order through a topological sort of the tasks in the **depends_on** relation

input: the error to propagate
$blamed\_tasks = \{t_1, t_2, \ldots\}$ the set of tasks associated with the error
$handling\_tasks = \{h_1, h_2, \ldots\}$ the set of tasks in the plan so that:

1. h_i or the planning task of h_i is able to handle the error which is being propagated.
2. there is at least one element of $blamed\_tasks$ which can be reached from h_i through the **depends_on** relation.

if $handling\_tasks = \emptyset$ **then**
    the error is not handled
else if $handling\_tasks = \{h\}$ **then**
    only one candidate, call the error handler defined by h or its planning task
else
    Let h be an element of $handling\_tasks$ so that all other tasks of handling_tasks are parents of h through **depends_on**
    if h *exists* **then**
        call the error handler defined by h or its planning task
    else
        multiple candidates, call decision control to handle the situation
    end
end

than one task can handle the error is still an open problem in our system: exception propagation has no choice but to call the decision control component.

Finally, note that since planning tasks are involved in the exception propagation, integrating the repair capabilities of a planner is simply done by calling the planner when exceptions are caught by the corresponding planning task.

3.2.3 Handling remaining errors

After this error handling phase, the system acts upon the following remaining errors:

- all errors found during the execution phase of the cycle that have not been handled. These errors are the errors related to exceptions thrown by user code.
- the errors returned by a second pass of the global plan analysis procedures. This second pass is there to make sure that the exception handlers did repair the plan.

At this stage, it is not possible to repair or handle these errors. The system marks the involved tasks for garbage collection, so that they are interrupted and removed from the plan:

- if the failure point is a task or a task event, we mark the task event and all tasks that are parent of this task through *any* task relation graph.
- if the failure point is an event outside a task, we mark *all* tasks linked to this event through *any* event relation graph (child and parents).

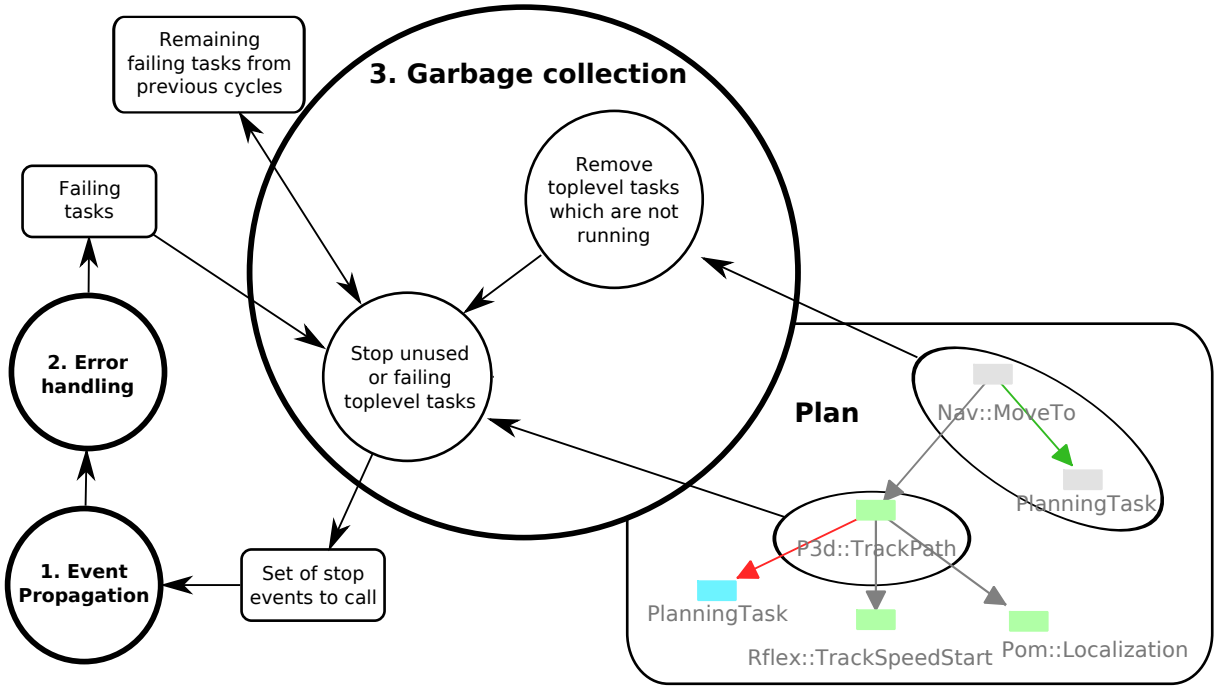


Figure 3.10: Overview of the garbage collection cycle: the unused or failing toplevel tasks that are still running are killed and removed only when they are finished.

This scheme can be summarized as follows: if an error has been found, and if nothing handled it, then kill all activities which are involved in this error. We simply assume that the plan as it is is broken, and that all possible ways to handle the problems have already failed.

3.3 Garbage collection

The last stage of the execution cycle, the garbage collection cycle, is to detect and remove tasks that are either no longer needed by the plan or cannot be kept because they are involved in an unresolved error. Unlike in other hierarchical task-based systems, we do not expect each task to handle that for its children: doing it in a separate phase, in which the plan is analyzed globally, allows more flexibility. For instance, tasks which share a common child do *not* have to care about when to kill that child. Another example of its usefulness is that an activity management strategy can be based on global concerns: for instance, a task can be kept because a planner claims that it will be using it in the plan it is currently generating or because a remote plan manager claims that it depends on it.

Fig. 3.10 presents an overview of the garbage collection phase.

3.3.1 Useful tasks

As already mentioned, task relations express a notion of usefulness: in a $t_1 \rightarrow t_2$ relation, t_2 is useful to t_1. This is a central notion for garbage collection as it allows to trace the usefulness of each task in the plan: given a set of tasks that are marked as useful *by definition* – see below –

the system computes the set of tasks that are useful for the system as a whole. The tasks that are "garbage" are then remaining tasks.

The following tasks are always useful:

- *missions*, which are the tasks which are the current high-level goals of the robot.
- *permanent tasks*, which are a set of tasks that should not be automatically garbage-collected. This is used mainly to start some common services (like localization) once and for all and keep them running.
- *tasks used by plans being built.* Our plan manager represents the plans that are currently being built, allowing to keep the tasks that are not immediately useful in the current plan, but may become useful in the future evolutions of that plan.
- the garbage collection of tasks involving other plan managers is presented in the next section, dedicated to multi-robot plan execution.

The tasks that have been marked by the error handling phase are always included in the set of tasks to be killed.

3.3.2 Killing the tasks

The second part of the garbage collection phase is to actually kill the tasks. To do that, the set of tasks that are *immediately* unneeded is computed: in the set T_u of tasks just computed, some of them still have parents which are in T_u as well. It is therefore not possible to stop these tasks since they are still useful for their parents and killing them now could break the termination process of their parents. Therefore, at this stage, the only tasks that are killed are the tasks in T_u that have no parents.

Among those, the garbage collection scheme removes the tasks that are not started and terminates the tasks for which it is possible: the ones for which the e_{stop} event is controllable. Moreover, the following cases must be taken into account during garbage collection:

- the task is being started: its e_{start} event has been called but is not emitted yet. In that case, garbage collection must wait for it to be emitted before calling e_{stop}.
- the task is starting, but its e_{start} event failed to emit: either a `CommandFailedError` or an `EmissionFailedError` have been found for that event.
- e_{stop} is controllable, has been called but fails to emit: either a `CommandFailedError` or an `EmissionFailedError` have been found for that event.

For the second and third items, the plan manager places these tasks in a quarantine zone where they are kept. The plan manager itself can do nothing more with these tasks – it does not try to stop them again for instance.

Note that shutting down the plan manager is simply marking *all* tasks for garbage collection. The plan manager is then shut down when the quarantine zone is empty and all removable tasks are removed (i.e. e_{stop} is not controllable on the remaining tasks).

3.4 Distributed execution

The execution of joint plans – plans which involve more than one robot – requires of course some specific mechanisms. We present here how our plan managers executes multi-robot plans in an environment where communication is not always available. What our system achieves is to have the "distributed" part of the execution being completely transparent, while taking into account the specificities of multi-robot execution. In that regard FIRE [66], an extension of TDL for multi robots, and PRS as used in multi-robot [4] do not consider multi-robot systems as being specific: FIRE for instance allows to develop TDL's task trees among multiple controllers but does not handles problems linked to the asynchronous nature of robot-to-robot communication. COMETS [32; 31], on the other hand, has been designed with multi-robot in mind. However, it has been done by completely removing the management of joint tasks from the "central" plan. Joint tasks are completely managed by a separate component and do not share the same task model than the central planner. This separation of models and tools contrasts with our goal to provide a central tool for plan management. Finally, the TAEMS task model and the tools built around it does handle execution of multi-robot plans. However, the TAEMS task model does not represent joint tasks activities, and thus does not allow to manage a teamwork model around TAEMS plans, something which is needed in our opinion.

This section first describes the communication protocol used between plan managers. Then, it shows how to handle events in a multi-robot plan. Finally, it discusses the fact that the transparency we are aiming at is not perfect: there are behaviour differences between the execution of a local-only plan and a joint plan.

3.4.1 Communication with other plan managers

In order for our system to handle the loss of communication, we have to define two distinct notions to manage the communication with another plan manager:

connection two plan managers are *connected* if they are involved with each other. Upon connection, a `ConnectionTask` task instance is created and started in the plan. This task instance is terminated when the connection is closed or when a protocol error occurs (i.e. if there is a bug in the communication layer).

communication link the communication link is either *alive* if it is possible to send data to the peer, or *dead* otherwise.

The notion of *communication* represents the physical communication link between to plan managers, while the notion of *connection* represents the fact that two plan managers have accepted to work together. One could see it as a logical connection between the two plan managers. For instance, if two peers break their connection with one another, then both can assume that their joint tasks are broken.

Then, the data flow between two plan managers is based on a set of messages which describe:

- execution-related changes: such as event emission, signal or forwarding between a local event and a remote event.

- changes of attributes such as added and removed owners for a task.
- structure-related changes such as added and removed tasks, added and removed relations.
- barriers, which is a specific message which stops message processing between a given set of hosts until all hosts have processed the corresponding barrier message. This is needed when synchronization points have to be set among multiple plan managers: a barrier guarantees that any message received after the barrier would be processed only when *all* involved hosts have processed *all* messages received before it.

Of course, a plan manager is only notified of events which involve either one of its own objects or one of the objects it is subscribed to. Moreover, we do not resend messages received from other plan managers. This is to avoid having the same message received from multiple different routes by a plan manager, which is a hard problem to solve and is outside the scope of this thesis.

Because of the structure of our execution engine, we consider that all messages generated during one cycle should be sent as a whole. Each plan manager gathers this set of messages and processes them during the first phase of its execution cycle, while keeping the message order. More specifically, given a set of messages $\{e_1, m_2, e_3, \ldots, e_i, m_{i+1}, \ldots\}$ where e_i is an event-related message (emission, signal or forward) and m_i another kind of message, we process m_i only after all e_j have been emitted so that $j < i$. This guarantees that a non-event operation is done only *after* all events that might have been its cause.

Causality, however, is not guaranteed between events which come from different plan managers: assume there is a chain of events $a \xrightarrow{sig} b \xrightarrow{sig} c$ where each event is owned by a different plan manager, and that the plan manager of c is subscribed to both a and b. The notification of $a \xrightarrow{sig} b$ will come from P_a and the notification of $b \xrightarrow{sig} c$ will come from P_b. Since there are no guarantee of ordering there, it is possible that P_c is notified of $b \xrightarrow{sig} c$ before it is notified of $a \xrightarrow{sig} b$. This is a limitation of the current implementation, but it can be solved in future evolutions by keeping track of the causality between the messages of one system and the messages of another system.

3.4.2 Handling joint events

Events of joint tasks cannot be handled as local events are: since these "joint events" are to be handled by more than one plan manager, we have to put into place rules that guarantee synchronization between the involved plan managers. The following rules are in effect:

1. when the command of a joint event is called, it is called on every plan managers owning this event. The command which is called can be role-specific or do nothing, but all plan managers that are involved in the management of this event must be called.
2. the event is emitted only when *all* the owning plan managers declare they are ready to emit it.

The notification mechanism described in the previous section guarantees the application of the first rule: a plan manager, when signalling the joint event, will notify all its owners of this

signal. The second rule is, however, truly multi-robot specific. It is implemented by electing a *master* among the owning managers when the event is being emitted. This master is notified when the remaining owners are ready to emit the event. When all owners have done so, the master emits the event. Since the other managers are also subscribed to the joint event, they are notified of the emission in the normal way. Note that this mechanism is not a specific one: a mono-robot task being, in our plan, a task with a single owner, the joint tasks and mono-robot tasks are not treated differently by the executive.

This handling of joint events is a representation of the establishment of a mutual belief in the joint action theory of Cohen and Levesque [22]: when a robot believes that it must start a task ("goal" in the joint action theory), the system informs the other robots of that fact and the robot can start its action only when *all* the involved robots have (i) been informed that the joint action is to be executed and (ii) accepted to do their part. The emission of the e_{start} event of a task is therefore the equivalent of the establishment of the realization of that task as a joint persistent goal:

- all the robot mutually believe that the task is not yet achieved.
- they all accepted to make the task achieved (i.e. they *want* the task to be achieved).
- they will continue to believe so until they have reached the belief that either the task is achieved or it is not achievable.

The rest of the information transmitted about the joint tasks helps to share the robot's beliefs about the task possibility of success. For instance, if a child task fails, the other robots will be notified as soon as possible of this fact: the robots which are affected by it will be eventually notified of the e_{failed} event emission.

3.4.3 Behaviour differences between local plans and mixed plans

Some behaviours differ between local plans – in which there are tasks from only one plan manager – and mixed plans. At the frontier between the plans of two different managers:

- the propagation of $a \xrightarrow{fwd} b$ is no more synchronous when b is handled by a plan manager M_b and a by another plan manager M_a. b can be emitted in M_b only if M_a notifies that it is, so there is at least one execution cycle between the emission of a and the emission of b. Even more so in a chain $a \xrightarrow{fwd} b \xrightarrow{fwd} c$ when c is also handled by M_a.
- exception handling does not allow plan reparation to cross the borders of each plan manager: an exception generated by a constraint violation in one plan manager will not be passed to another one. Moreover, exception handling is a synchronous process (the plan must be repaired "just after" the exception propagation phase) and our plan manager is built with the idea that communication links are not always available. The only generic way to specify error handling as one robot repairing for another robot is therefore plan repairs.

Moreover, there is an intrinsic asynchronicity between event propagation and plan changes: a remote plan manager can always remove a signal while we have already queued its emission.

Such a situation is not an error in the communication protocol. It is simply an error in the plan: either the remote peer broke the plan because of some error or critical situation (it decided to drop the joint plan), or there is a lack of synchronization which led to this fault. This error is handled as a normal error, through a `RemoteEmissionError` which is associated with the source events of the signalling or forwarding operation.

Finally, the garbage collection mechanism must take into account the fact that the plan includes tasks which are useful for other plan managers. If the plan contains a remote task t_r, then any local task t_l which is linked to t_r are marked as useful. This is done because, in the system, the plan is used as a form of weak contract: when the plan manager allowed the $t_r \rightarrow t_l$ relation to be added, it announced that it would try to achieve t_l for the remote peer(s). Therefore, t_l cannot be automatically removed.

3.5 Summary

This chapter has presented the different processes defined for plan execution. The execution cycle is divided into three processes:

- the *propagation phase* in which the system reacts to external events according to its plan. This phase involves a pseudo-synchronous propagation algorithms which enforces a partial ordering of events. During this phase, the application code defined for the event commands and the events handlers is called.
- the *error handling phase* offers a way to handle the various detected errors. Our system classifies the errors into two main categories: *code errors* which have been raised by event handlers or event commands, and the *constraint violations* which are detected by global plan analysis procedures.
- the *garbage collection phase* which kills and removes the tasks which are not useful anymore for our system.

As we stated in the first section, the event propagation phase is a necessity for a system based on discrete events like ours. The main contributions of our plan execution scheme is the implementation of multiple error handling schemes and the garbage collection process.

Our system offers multiple ways to handle errors. *Simple errors* can be recovered by modifying the main plan directly in event handlers. As this is done in conditional planning, the way to recover from *common errors* can be defined directly in the plan through plan repairs and the **error_handling** relations. Finally, an *exception* mechanism allows to integrate non-local repairs: a high-level task can repair a low-level task by defining the corresponding exception handling procedure. This integration of these three main error handling schemes allows to use the right scheme for the right situation, and keep the error handling part of the plan and of the application code simple.

By having a separate garbage collection phase to remove unused task, we remove the burden to determine which tasks are unused and how to stop them from the developer which write the task models. Moreover, since our system represents task *graphs* and partially built plans, the

determination of the set of unused tasks truly requires a global analysis procedure like the one we implemented.

Finally, during plan execution, the decision control component is used to arbitrate between multiple ways to handle errors:

- choose between multiple matching **error_handling** relations.
- choose between multiple candidate exception handlers.

Our execution model is inherently multi-robot: event propagation and the constraints defined by task relations, combined with the notion of ownership, allows multi-robot execution. The only multi-robot specific addition is the handling of joint events, which are used as synchronization point: they can be emitted only if all the involved robots agreed to their occurence.

The next chapter will present how plans can be safely adapted while they are being executed, and what operations our plan management component offers to transform plans during their execution.

4

Plan Management

The previous two chapters have presented what is the model used to describe plan in our plan manager and how a plan is executed once it has been built. This chapter ties the two things together by explaining the tools offered to change plans and to change them while they are being executed.

Most supervision systems allow some kind of online plan modification: either because an error makes the current development backtrack and develop another possible branch (PRS and TDL behave this way), or because they are trying to instantiate a plan which is better with respect to the robot updated beliefs (Structured Reactive Controllers, TAEMS). Contingent planning is also another instance of online plan modification strategy. In our plan manager, a central tool is used to support the safe concurrent modification and execution of plans: *transactions*.

The first section presents these transactions, how they work and discusses what they achieve (section 4.1). This is a central tool in our system, used to change the plan being executed and to negotiate on plans, in particular in multi-robot context. It is also central in that it allows to handle the conflicts which may appear between the changes brought by execution and the changes described by the transaction.

Then section 4.2 describes two basic modification operators, and more importantly how these adaptation operators interact with execution. These two modification operators are example of what can be built upon our plan model and our execution scheme.

Finally, section 4.3 will present a short summary of the concepts implemented in our plan management component to adapt plans as they are being executed.

4.1 Simultaneous plan modification and execution

None of the systems we just mentioned really deal with the issue of preparing modifications for a plan which is currently being executed. Beetz, for instance, considers cases where a planner builds a new improved plan while the current one is being executed. He does not, however, considers that the plan being built can become incompatible with the execution, how to represent this conflict and how to react. We will show in this chapter that our system provides such a mechanism.

In the Claraty architecture, this issue is dealt with by using a "floating line" which separates the part given to the executive, which is in fact being executed, and the part given to the planner, which can be modified. This does not really fix the problem, as failures on the short-term plan can impact the plans that are being built by the high level planner. This is not really an issue if the planning times are short with respect to the dynamics of the system, and as such it is a successful scheme in mono-robot context. But in the case of multi-robot systems, communication latencies can lead to long negotiation and robots need to consider the execution status and how this status evolves while they are building a joint plan.

4.1.1 Motivation

Our system does not have the model needed to check that the robot state is compatible with starting a task (for instance): our plan manager relies on the task and event relations to assess what is broken and what is not. The executable should therefore not consider a plan in which relations are not yet established and/or tasks are missing for execution. Note that such a "sound plan" can be a non-executable plan: it is possible that a plan includes tasks that are either abstract or partially instantiated: in that case, this non-executable plan is sound as it gives the information needed to develop a fully executable plan which is coherent.

The golden rule of plan management in our system is therefore the following:

Rule
The plan which is seen by the executive, also called the "executed plan" or the *main plan*, is always sound as long as the plan generators build sound plans.

It is easy to see that this rule cannot be enforced if the main plan is concurrently modified and executed: at some point there would be some task, event or relation missing. We need something to represent a plan change outside of the main plan, and we need to be able to apply these changes all at once to the main plan.

Note that such a scheme is also required for some of the mechanisms already presented to work as expected: if one was to add a single task in the plan, this task would most likely be garbage-collected. Moreover, we can see on Fig. 4.1 that not having this tool could break the notion of trigger.

Our plan manager provides the *transaction* as the mechanism to modify plans. Transactions are already common in the database world: they represent a set of modifications to a database,

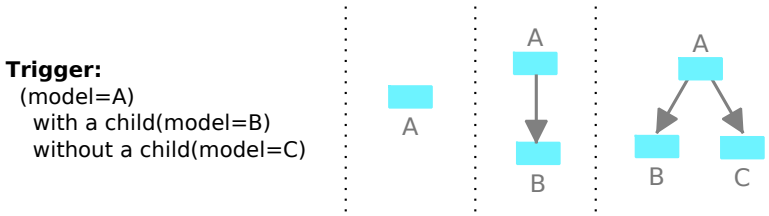


Figure 4.1: how incrementally changing the main plan can break triggers: A matches the trigger in the second step, while it should not since the final "sound" plan is the third step and the trigger does not match here

guaranteeing that these modifications are applied either all at once – in which case the transaction is said to be *committed* – or not at all – in which case it is *discarded*. This idea has been adapted to the management of plans.

First, we present how transactions represent a set of changes to plans. Then, we show that they also provide a basis of negotiation in multi-robot context. Finally, the interactions between the plan which is being built and the changes that are done to the main plan because of its execution are described.

4.1.2 Representing plan modifications

Transactions are represented as a new state of part of the plan: they tell how, all other things being equal, a set of tasks and events should have their relations set to form a new sound plan. This differs from Beetz's approach to plan transformation: in Structured Reactive Controllers, the plan model expresses what transformation to apply and when to apply it. In our case, we do not want to provide predefined models of plan modifications: we want, as a new plan is being formed, to be able to track how it changes the main plan and make the executive and planning systems interact. Then, any part of our architecture (decision control, plan generation or task code) can change the plan when needed. The parts "what to change" and "when needed" are not represented in our plan manager.

To better describe how transactions work, let's take a simple example: the repair of the `P3d::TrackPath` task already presented earlier (Fig. 4.2). This transaction represents the change from the main plan in which the task failed into a repaired plan. We see that transactions contain two kind of tasks: the tasks that are already in the main plan (white and blue tasks) and the ones that are being added (blue tasks).

A transaction is therefore defined by a tuple $(P, O_{new}, O_{removed}, O_{proxies}, R)$, where:

- P is the plan the transaction applies on. In our example, it is the main plan, i.e. the plan being executed, but it can also be another transaction.
- O_{new} is a set of tasks and events that are not in P. In our example it is `P3d::TrackPath` and a new `Rflex::TrackSpeedStart`.
- $O_{removed}$ is a set of tasks and events that are in P but should not be in the new plan. It is usually empty as task removal is handled by the garbage collection pass: the tasks that

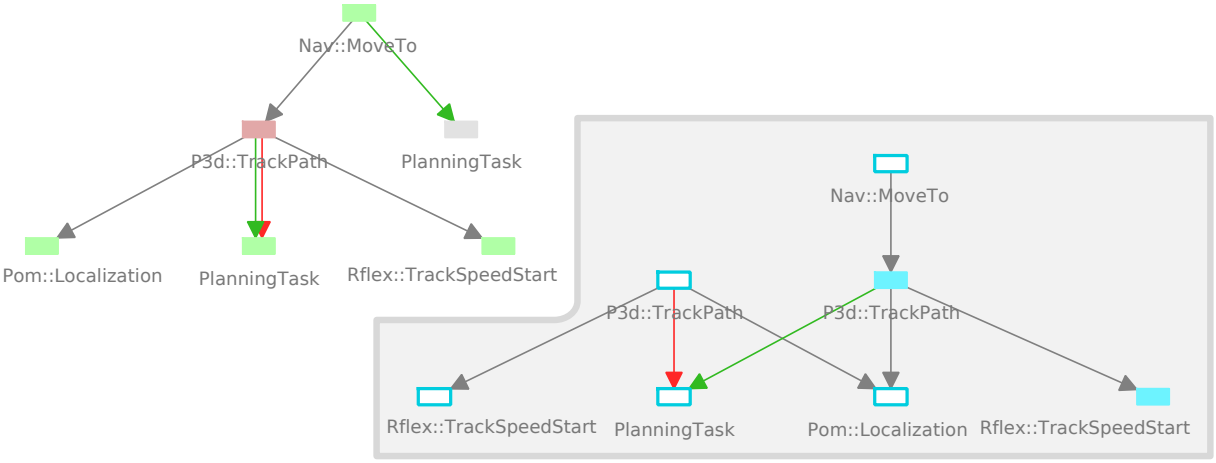


Figure 4.2: Example of a transaction: in the plan being executed, on the left, `P3d::TrackPath` has failed (red). On the right, `PlanningTask` is using a transaction to generate a new valid plan. The tasks in white and blue are *transaction proxies*: placeholders for tasks in the main plan which are affected by the transaction.

are not useful for the new plan are removed automatically.

- $O_{proxies}$ is a set of tasks that are in the plan and have a representation in the transaction. In our example, this is the set of tasks that are represented in blue and white.
- R is a set of relations between the objects in $O_{new} \cup O_{proxies}$. This defines what should be *the* set of relations between these objects in the new plan. In our example, we see that, for instance, there is no relation between `Nav::MoveTo` and the old path tracking task but there is one between `Nav::MoveTo` and the new one. The new plan should have these modifications as well.

A tasks which is in neither $O_{removed}$ nor in $O_{proxies}$ is not modified by the transaction. A transaction can also change the *attributes* of a task: set argument values for partially instantiated tasks, set a task as mission or remove it from the set of missions, add or remove owners, ... Virtually anything which defines a task can be changed in a transaction.

Once a transaction is finished from the point of view of the plan generation components, it can be applied to the plan. This operation is called a *commit* and it is done by changing the relations, adding new tasks and removing the tasks that should be. The tasks that are left over after the commit are then removed by the garbage collection pass (Fig. 4.3). At any time, the transaction can be *discarded*: it is simply removed from the list of active transactions.

4.1.3 Conflicts between execution and planning: transaction edition cycle

As transactions represent a change of the main plan, it is easy to track how the changes in the execution of the main plan affects the new plan which is being built. This section deals with that issue: what execution-related changes makes a transaction invalid and how these situations are handled in our architecture. In particular, we will see that the decision control component has a central role: it acts as a mediator between execution and planning.

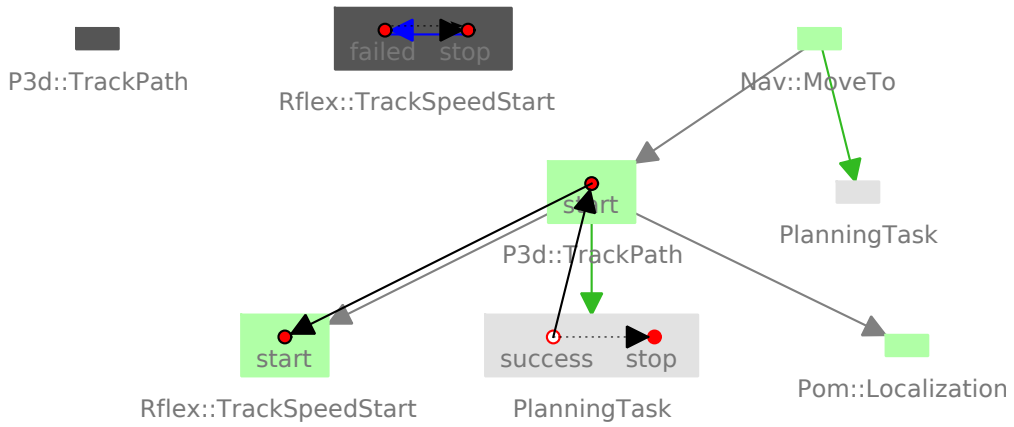


Figure 4.3: Result of the transaction on Fig. 4.2 when it is committed. The plan manager garbage-collected the old `Rflex::TrackSpeedStart` since it is not marked as useful anymore

This problem is for instance solved in Claraty by freezing the short-term plan: the designer assume that, by forbidding the change of the short-term plan, no conflict will ever arise. This is a problem in general for obvious reasons:

- one cannot modify the short term plan as you modify the long-term one.
- planning is hardly a time-bounded process, and as such it is difficult to *know* where to put the limit between "long term" and "short term".
- the plan being built can depend on activities already on the short-term one (for instance, localization). So, when the new plan is built, it already depends on the outcome of the short-term part of the plan. Freezing the short-term plan did not really solve the problem.

Conflict sources are multiple: Fig. 4.1 lists the conflicts currently detected by our system. An interesting case is the "event called" case: the plan execution lead to calling an event whose emission breaks the plan structure in the transaction. It is interesting since this is a controllable action: in our plan manager, this case is detected *before* the event is called and decision control can choose to favor the transaction over the main plan. In this case, the transaction is still valid and `EmissionFailedError` is generated in the main plan (cf. section 3.2.1). Note that handling situations where, for instance, generating the `EmissionFailedError` would also break the transaction is to be considered by decision control. We therefore have two kind of conflicts:

- contingent conflicts, which the execution engine cannot avoid. In that case, the only option is to change the transaction. An example is the violation of a constraint induced by the emission of a contingent event.
- controllable conflicts, which it is possible to solve at the execution level.

An obvious interesting extension to this conflict management would be to compare transactions one with the others, and fix potential conflicts which would appear if one transaction was to be committed. We did not investigate this yet though.

To support the communication process between the execution, the planner and the decision control component, we now define the *edition cycle*: from the point of view of the executive,

Type	Description	Possible resolution
Relation change	• a relation in R has been removed from the plan. • a relation has been added in the plan between two objects in the transaction, and this relation is not in R.	• make R match the plan: add or remove from the transaction the corresponding relation. • acknowledge the difference between the transaction and the plan (i.e. accept the new plan as the new basis for the transaction).
Constraint violation	A constraint induced by a task or event relation is violated in the transaction, possibly because of an event emission	Change the transaction so that there is no constraint violation anymore in it.
	An event, whose emission would break a constraint violation, is called in the plan	• change the transaction. • do not call the event.

Table 4.1: Possible conflicts between execution and transactions. While the list of conflicts is extensive given the current implementation of our system, the possible resolutions are only examples.

transaction management is mostly asynchronous. Only controllable conflicts require decision control to act as a mediator. Note that the corresponding decision must be taken quickly with respect to the system execution cycle length, since the execution engine waits for decision control.

The planner must take into account the conflicts. Editing the transaction must be done in a cycle in which plan modification phases is interleaved with communication with the decision control component. The overall communication inside our architecture is depicted on Fig. 4.4. When a conflict appears, the transaction is immediately marked as invalid and remains so until the conflict is solved. While the transaction is invalid, it cannot be committed. Note that even if this process requires the planner to have some kind of planning/interaction cycle, we do not lose generality here: (1) most planners do have a "main loop" in which this cycle can be implemented and (2) if the planner cannot or should not be modified, this cycle can be implemented inside the plan manager itself, it would just be less efficient than (1).

4.1.4 Transactions as distributed whiteboards

Transactions are distributed objects: they can be used to build joint plans and more importantly to negotiate on the structure of these plans. We will see in this section the specifics of building multi-robot plans using transactions.

First, let's review the edition cycle just presented above. In multi-robot plans, more than one robot modifies the transaction, so we need to explicitly express who is modifying the transaction. Moreover, as we already mentioned, we want the resulting plan to be a contract between the involved robots. Therefore, the *transaction commit* should be a way for the robots to *sign* this contract: it guarantees that the transaction is committed in *all* involved plans or not at all.

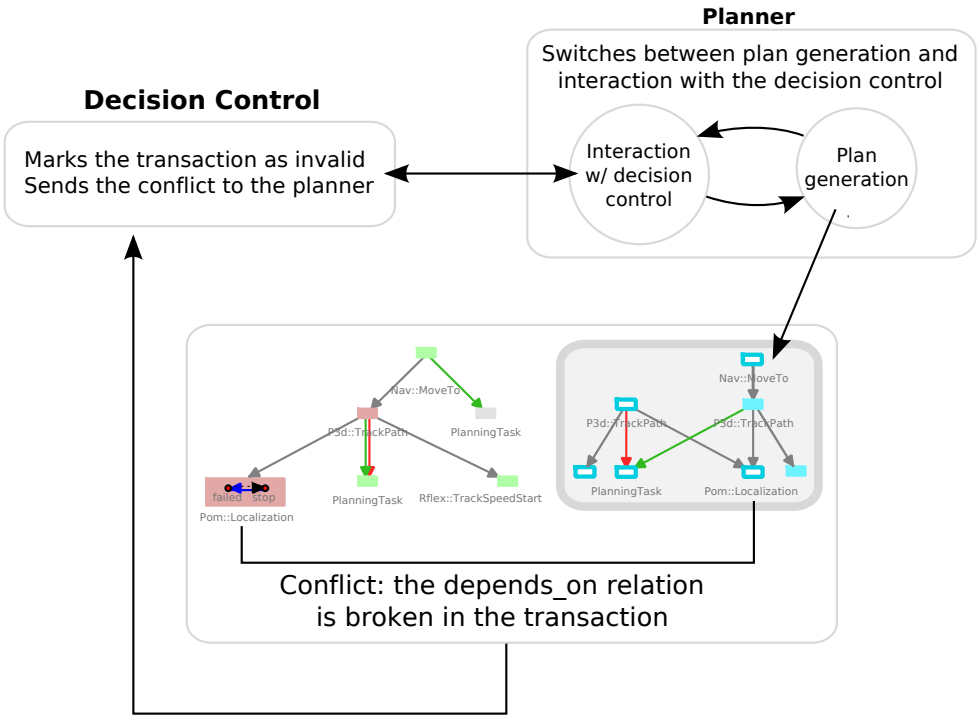


Figure 4.4: the process of transaction edition: how the conflicts detected by the execution engine are transmitted to decision control, and from there to the planner. The planner then receives that information during an *interaction* phase and adapts the transaction (or discards it) during a *planning* phase. The modification of the transaction can be done asynchronously during the *planning* phase.

For these reasons, we added an *ownership* attribute to transactions, which lists the plan managers which are allowed to modify it, and the edition cycle is changed as follows: the transition between the two states of the planners (Fig. 4.4) now depends on the availability of an edition token. This edition token is passed among the owners of the transaction in a token-ring protocol. Moreover, the token keeps two informations: who changed the transaction and who did not change it, and who asks for the token back again once. A transaction can be committed only when the token has been passed once among all the owners but none changed the transaction, and none asked for the token again: when that happens, we can indeed assume that everybody agrees on the transaction result.

The transaction commit is then done in two steps:

1. a "prepare to commit" message is sent to all peers, which can accept it or not. If one of them refuses it, the transaction is discarded.
2. if all plan managers accepted the previous message, a "commit" message is sent, followed by a barrier (cf. section 3.4.1). All plan managers *must* commit. If one plan manager fails its commit, an recovery process ensures that all tasks that have been changed by the commit are killed.

The first step has been added so that the processing of the commit message is as fast as possible, reducing the probability that a commit recovery is needed. This is a basic flaw in distributed commits: there is no tool, as for now, to properly recover from a failed commit.

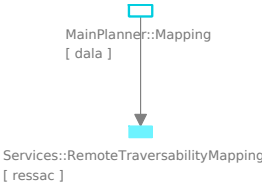
In addition to this changed edition process, the following specificities are to be taken into account when plans are built into distributed transactions:

- *subscription*: if the default mechanism of automatic subscription is not enough (see section 2.3), the planners must add explicit subscription to the plan. Note that, while the transaction is being built, there is no "partial view" of the transaction: everybody sees all tasks that are added to the transaction.
- *ownership*: adding tasks to the transaction is not enough, the planners must also properly set the ownership attribute.

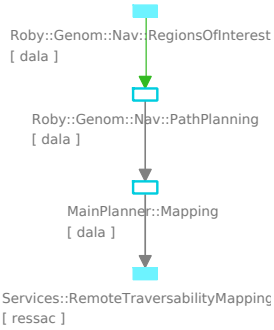
Distributed transactions are the central tool in which plan-based negotiation can take place. In our experiment, the joint plan of the UAV and the rover is of course built using a distributed transaction (Fig. 4.5). They are not a negociation protocol but a basis for the development of such protocols – and for the integration of already existing ones.

4.2 Modifying plans

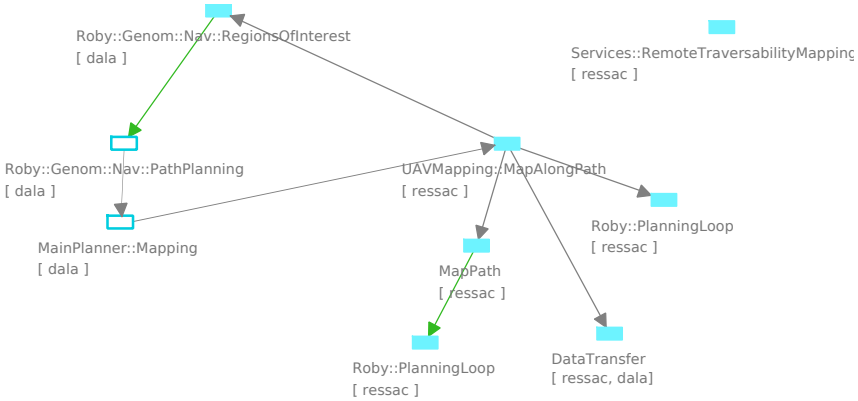
Now that plans can be modified either synchronously online or asynchronously by using transactions, we will discuss the tools of plan adaptation: first, the online modifications of plans in multi-robot context are presented. Then basic plan modification operators are described. Finally, the plan merging system, an online plan adaptation scheme built within our plan manager, is described. This last section will show the benefits of centralizing the plan representation: once this plan merging is in place, it can adapt all plans regardless of what planner generated it.



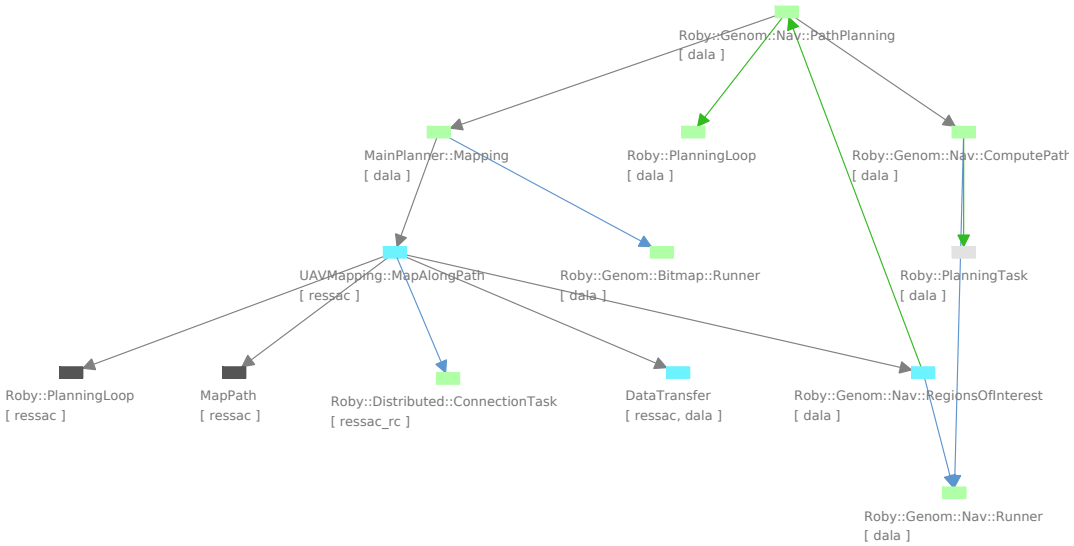
(a) The Ressac UAV is notified of the presence of a `Services::TraversabilityMapping` task in the rover plan. It builds a transaction which at this point expresses that it can help the rover through the use of an abstract `Services::RemoteTraversabilityMapping` task. The transaction is sent to the rover.



(b) The rover accepts the transaction and creates a `RegionOfInterest` task representing its ability to generate a set of regions to be perceived by the UAV.



(c) Based on the available information, the UAV chooses its mapping modality (`MapAlongPath`) and replaces the abstract `Services::RemoteTraversabilityMapping` by it. It generates the rest of the plan. The transaction can be committed.



(d) Final plan from the point of view of the rover: the tasks in black are the tasks removed because they are not useful for the rover's plan. They would have been kept if the rover explicitly subscribed to them.

Figure 4.5: Building the rover/UAV joint plan through a distributed transaction

These operators may be the very reason why supervision systems like our plan manager, separated from the planning systems, are useful from an engineering point of view: they can properly handle the specifics of execution-related problems and let the planners reason on more high-level models.

4.2.1 Ownership and online plan modification

In our system, plans describe who is doing what through the ownership attribute. Allowing a plan manager to change the plan as it sees fit would therefore be equivalent to having let any plan manager have authority on all the other plan managers. This is something we want to avoid in our system:

Rule
A plan manager can *add* activities or activity relations on the plan of others only through negotiation. Moreover, it cannot add an event relation pointing to the plan of another manager.

Note that if a plan manager adds an event relation *pointing to* one of its local event, it simply asks for notification – and it could be done "by hand" using subscription. This modification is can therefore be done freely.

Because of this rule, the only way to change plans without restriction is to use transactions: a plan change is only allowed if all involved plan managers commit the transaction: negotiation is required in that context.[1]

However, online changes to a joint plan are not forbidden. As we already mentioned, the plan forms a contract between the involved plan managers, but this contract is weak and can be broken if needed:

Rule
A plan manager can remove its tasks from a joint plan at all times.

This means that a plan manager can always remove a relation if it is the only owner of one of the two objects involved in it. Moreover, it can remove itself from the list of owners of a joint task. In addition, this operation can be performed by the garbage collection phase: if a joint task is marked as being not useful for the robot – for instance because it has been removed from the robot's set of missions – garbage collection removes automatically the robot from the joint task's owners.

4.2.2 Switching plans

The most basic plan modification mechanism available in our plan manager is the modification of relations. In order to simplify the development of controllers, we defined more high-level

[1] note that having a master robot which can change the other plans freely is some kind of degenerate case where the other robots accept blindly the master's transactions

modification operators based on this idea: how to replace one task instance by another (or one subplan by another) during execution. The `replace_task` and `replace_plan` operators handle this.

4.2.2.1 Replacement operators

There are two operators to switch between task instances in plans, which have different uses: `replace_task` and `replace_plan`. `replace_plan` is closely related to the `SWAP_PLAN` operator of Structured Reactive Controllers, but adapted to the problematic of plan graphs. The next section discusses the case where the new subplan conflicts with the old one, and as such that the two tasks being swapped cannot run at the same time.

The motivation is as follows: given a plan and a task T in this plan, how can we keep the plan sound while exchanging the particular task instance of T with another instance T'. For instance, the Dala rover switches its motion modality by exchanging `P3d::TrackPath` and `NDD::TrackPath`. In order to perform this operation, we have to determine the following things:

compatibility of model is T' a valid replacement for T ?

compatibility of state if T is running, how can we make sure that T' is in an execution state equivalent to the one of T ?

These two questions are answered by the notion of *fulfilled model*: given a task T the fulfilled model of T is the model which is required by T's parents in the relation graphs. For instance, take the **depends_on** relation – in which a child is tagged with a $(model, arguments)$ pair the parent is depending on: T' can replace T only if it is compatible with all the models defined by all **depends_on** relations in which T is a child. Any task relation can define this kind of constraint, and the two replace operators first check the *compatibility of model* on this basis.

The *compatibility of state* is not directly handled by our plan manager. Each task model is supposed to provide a `make_state_compatible(`T`, `T'`)` routine whose job is to set up T' so that its execution state matches the one of T with respect to the model considered. The very fact that T is already in this state proves that the various models do not conflict with their notion of execution state (i.e. if T' matches the required model, then it can be brought to the right execution state since T *is* in this execution state). This method, however, does not apply if T' is used to replace T because T failed. `make_state_compatible(`T`, `T'`)` must also be able to make T' match the execution state of T *in the past*: in the case of a failed task, we specify that the execution state we want to reach is the one that T had *before* a certain event (which in a case of a failed task is the failure point of the error). The default behaviour (i.e. the `make_state_compatible` routine of `Roby::Task`) simply starts T' if T is running or failed. Note that the replacement may not be instantaneous, and may not be simple if it is impossible to have T and T' running together. The handling of this *transition* phase is described in the next section.

Once the two tasks are in a compatible state, one can apply the two plan modification operators defined for task replacement:

Definition

`replace_task(`T`, `T'`)` replaces T by T' in all task relations T is involved with. Moreover, all events of T are replaced by their equivalent in T' – if there is one – in all event relations they are part of.

The direct application of this operator is simply to replace one single task without modifying anything else. This operator allows, for instance, to simply restart a failed task when we know that the failure did not influence anything else in the plan. This is how we do the replacement of `Rflex::TrackSpeedStart` on Fig. 3.8 (section 3.2.2.1 page 65).

However, this operator is obviously not applicable in the modality switch we described: the two task implementations have different children, and it would be inconsistent to add the children of the old modality to the new modality. Another operator is used in this case:

Definition

`replace_plan(`T`, `T'`)` replaces T by T' in all task relations in which T is a child. An event e_T of T is replaced by its equivalent $e_{T'}$ in T' – if there is one – in an event relations $e_T \rightarrow e$ or $e \rightarrow e_T$ only when e is *not* an event of the subplan of T.
The subplan of T being defined as the set of tasks which can be reached from T through the task relation graphs.

The event side of `replace_plan` actually defines some kind of notion of *internal event relation*: we separate the event relations that are needed to manage the subplan of T and the ones that are used for synchronization with the rest of the plan.

Finally, we have to handle replacements done in a transaction: since transactions represent plan *changes*, we cannot allow tasks to be started inside a transaction (or the transaction cannot be discarded anymore). To handle replacements in that context, we record all replacements that have been performed in the transaction, and the relation changes related to them. When the transaction is committed, the following happens:

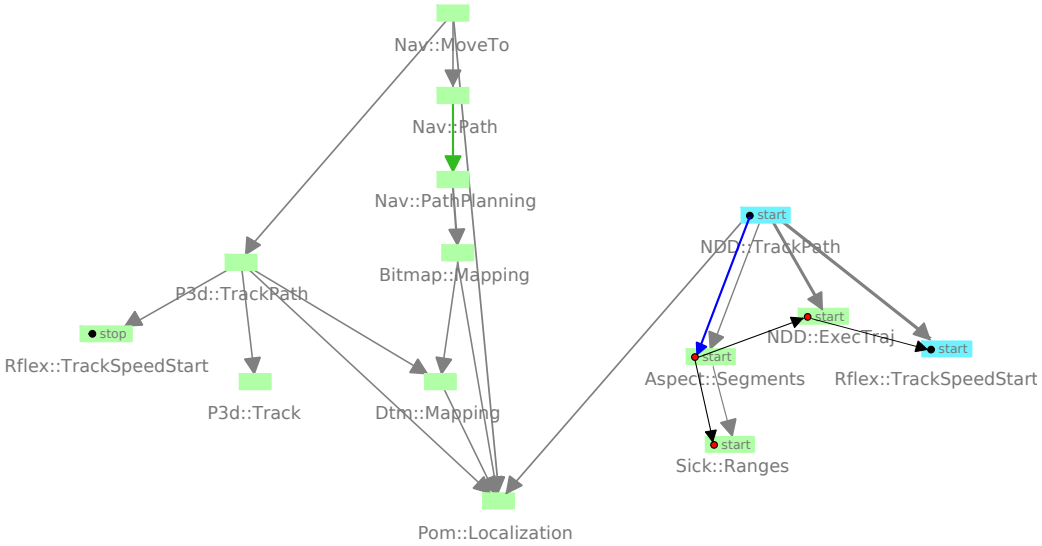
- the transaction is committed, but without the relations changes induced by replacements.
- the replacements are done as usual, thus taking into account the execution context.

4.2.2.2 Handling non-instantaneous task swapping: transitions

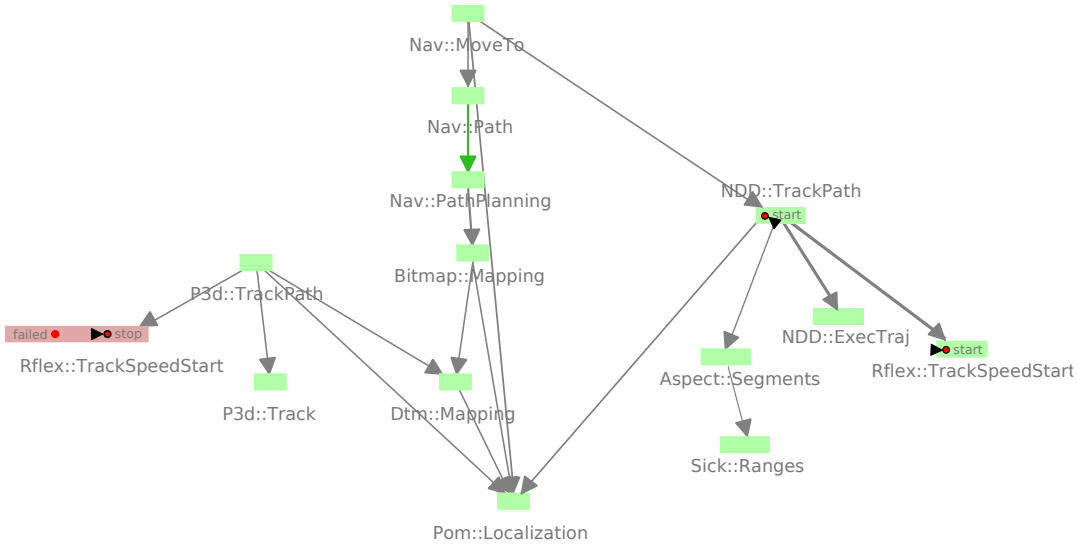
The problem with the swapping of tasks is that there is no guarantee that the new task (or the new subplan) can be started without some kind of conflict with the old one (Fig. 4.6). To handle that, we keep the set of task swapping which are in progress in the main plan, which is known as the list of transitions.

If a conflict is detected by the execution engine, the conflict is first compared with the set of active transitions. If a transition is found the following happens:

- the conflict is solved by giving priority to the task in the new plan. The old task is stopped and the new task is started when the old one has finished (Fig. 4.6(a) and 4.6(b)).



(a) The `NDD::MoveTo` task is brought to a state compatible with the one of `P3d::MoveTo`. During this process, the conflict is detected and the old `TrackSpeedStart` is stopped to solve it. The exception generated by the failing relation `P3d::TrackPath` and `Rflex::TrackSpeedStart` is inhibited while the transition is active.



(b) One the old `TrackSpeedStart` has finished, the initialization of the new motion modality continues. The old modality is then replaced and can be garbage-collected.

Figure 4.6: Switching motion modalities requires the use of a transition because the new modality conflicts with the old one.

- the exceptions raised because of this conflict resolution are inhibited while the new task is starting.

The transition is removed from the main plan set of transitions when one of the two following things happen:

- the new subplan fails to be put into the right execution state. The exception inhibition inserted if a conflict has occured is removed, and the normal error handling mechanisms are used.
- the new subplan is brought into the right execution state and the replace operation is finished (Fig. 4.6(b)).

4.2.3 Interrupting and resuming activities

In the plan manager, the interruption of activities is based on a `split` operator: interrupting a task is an operation which can split a subplan into two parts: the past in which the tasks are interrupted, and the future in which they have been resumed.

Let's consider a single task. The `split` operation is defined as follows:

Definition

Task modifications: $T_0, T_1 = split\_task(T)$ is an operation which is specific to the task T and returns two tasks so that T_0 represents the interrupted part of T and T_1 the resumable part of it. The e_{start} event of T_1 must therefore be controllable, but the e_{stop} event of T_0 can be contingent if the interruption process is not entirely controlled. Moreover, T_0 must be in the same execution state than T.

Event modifications: $split\_task(T)$ must transfer the semantic of T's events on T_0 and T_1 by changing the events relations. For instance, the e_{stop} event of T is equivalent to the e_{stop} event of T_1 but not to the one of T_0.

Default implementation: if the e_{stop} event of T is controllable, the default implementation of *split_task* is to create a new task T' of the same model with the same arguments. Trying to split a non-interruptible task is an error if there is no user-defined *split_task* operator defined for its model.

From this per-task `split_task` operator, we have to build a global `split` operator which handles task relationships: on the one hand, T has parents in the plan which depend on it and can therefore not continue their execution while T is interrupted. On the other hand, T's children can remain running since they should not depend directly on the fact that T is running. In a plan, the `split` operator is therefore defined as follows:

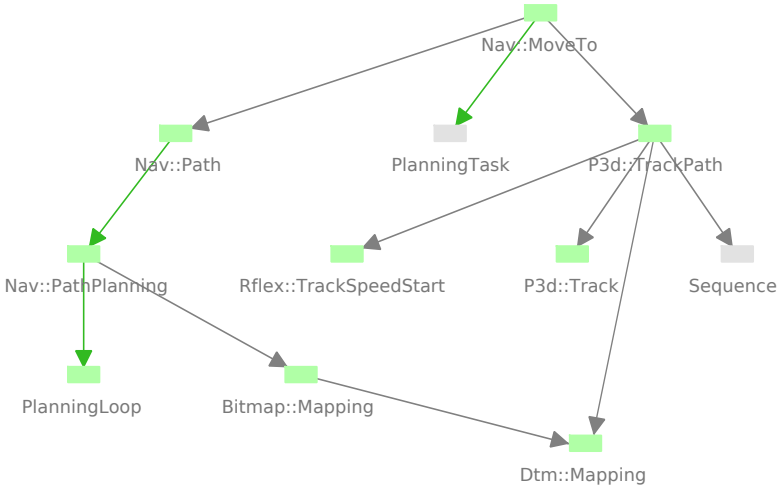


Figure 4.7: Starting point of the split of Fig. 4.8: a running `Nav::MoveTo` and its direct subplan.

Definition

Let $P_T = T_1, T_2, \ldots, T$ the set of tasks from which it is possible to reach T through the task relation graphs. This set is ordered topologically.

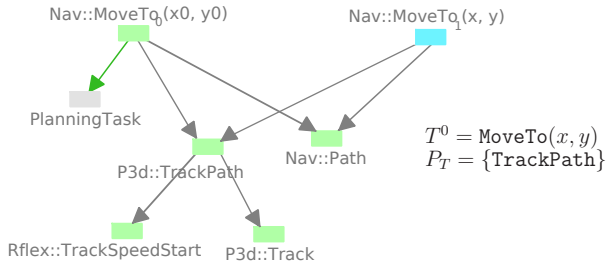
$e_{inter}, e_{resume} = split(T)$ is then defined by the algorithm 4.1. This algorithm is designed on the assumption that it is possible for $split\_task(T_i)$ to (i) queue some of its children for splitting and (ii) update the event graphs to specify some event orderings of interruption and resume.

See 4.8 for a broken-down example of how this algorithm works.

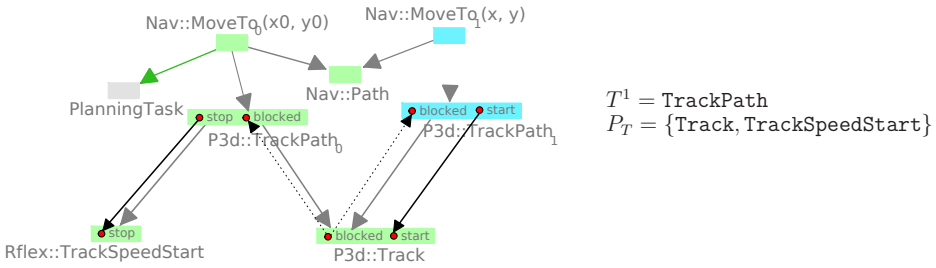
The behaviour of the *split* operator explains its name: the operator does not perform the operation itself (it does not stop the interruptible part of the new plan). It only transforms the plan into a plan in which the interruption is represented.

This *split* operation is central in the handling of conflicts: if the e_{start} event of a task T' is in conflict with an already running task T, one possible resolution is to interrupt T and start T'. This is actually a common pattern in plan reparation: let's assume that the localization task of our Dala rover fails. The error handler for this task is based on trying to get a centimetric GPS reading, something which is obviously in conflict with the movement of the robot, i.e. the `Rflex::TrackSpeedStart` task (Fig. 4.8).

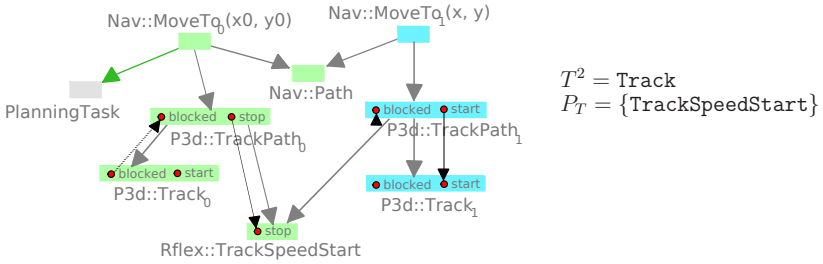
Now let's assume that the current position of the robot does not allow a relocalization. In that case, the robot would have to go back to a place where it knows it can read the GPS – for instance by going back on its tracks using both a local avoidance method and its known path. This would then conflict with *any* `Services::MoveTo` task in the system – in our case, the `Nav::MoveTo`. The current movement would have to be split and resumed later.



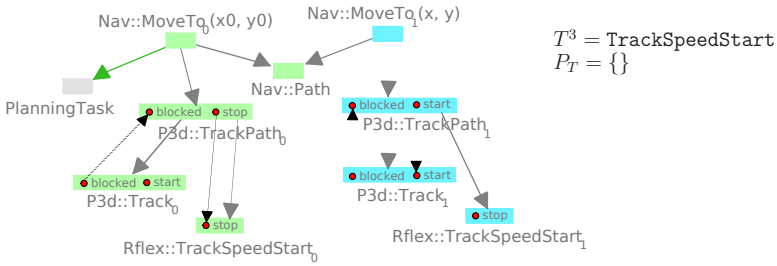
(a) The `Nav::MoveTo`'s implementation of `split_task` explicitly queues its child `P3d::TrackPath` for splitting since it wants the robot to stop. Moreover, the "past" part of the split has its (x, y) arguments updated to the current robot position.



(b) The `P3d::TrackPath`'s implementation of `split_task` explicitly queues its children for splitting for the same reason than `Nav::MoveTo`. Moreover, it updates the event relations to specify an interruption and resume ordering: `TrackPath`, which is the data source for `TrackSpeedStart`, should be stopped after it and restarted before it.



(c) The event relations set up by the previous stage are distributed onto $\texttt{Track}_0$ and $\texttt{Track}_1$ by the split algorithm.



(d) Final plan

Figure 4.8: Splitting operation for a `Nav::MoveTo` task. The initial plan is represented on Fig. 4.7

Algorithm 4.1: splitting a subplan into a part in which a specific task T is interrupted and a part in which T and the activities which depend on it are resumed. The algorithm recomputes the set of tasks to be handled at each loop: since a task can add its children to P_T, these new tasks can have other parents which should be split as well.

Input: the set P_T of tasks and the operation *split_task* as defined in section 4.2.3
Data: *Split* the set of tasks already split
Data: *involved_task*(S) returns the set of tasks which can reach any tasks in S through the task relation graphs
Data: *topological_sort*(S) sorts the task set S topologically, using the task relations graphs
Output: two event sequences e_{inter} and e_{resume}
let $i = 0$
while P_T *is not empty* **do**
  $P_T = topological\_sort(reachable\_tasks(P_T) - Split)$
  let $i = i + 1$, remove the first element T^i of P_T

  `// split` T^i
  $Split = Split \cup \{T_i\}$
  $T_0^i, T_1^i = split\_task(T^i)$
  add the e_{stop} event of T_0^i to the end of e_{inter}
  add the e_{start} event of T_1^i to the end of e_{resume}

  `// First, copy the relations internal to the split subplan`
  foreach $j < i$ **do**
    copy the task relations of $T_0^j \rightarrow T^i$ on (T_1^j, T_1^i)
    replace e_{T^i} by the corresponding event of T_1^i in all relations $e_{T_1^j} \leftrightarrow e_{T^i}$
  end
  `// Second, update the relations between the split subplan and the other tasks`
  foreach t_c *child of* T^i *in any task relation* **do**
    establish the same relations $T^i \rightarrow t_c$ on (T_1^i, t_c)
  end
  `/* the terminal events of` T^i `are equivalent to the terminal events of` T_1^i `and not the ones of` T_0^i `*/`
  foreach e_t *terminal event of* T^i **do**
    replace e_t by the corresponding event of T_1^i in all relations $e \leftrightarrow e_t$ if e is not an event of a $T^k, k < i$
  end
  do *replace_plan*(T_i, T_0^i)
end

4.3 Summary

This chapter has presented how our system allows to handle the problem of plan adaptation: (i) how plans can be simultaneously modified and executed through the use of transactions and (ii) examples of plan modification operators built upon our plan model.

Transactions are the central mechanism through which plans are built. Our system separates the main plan, which is the only one the executive can act on, and the plans which are in the process of being built, so that the executive always interprets sound plans. Transactions allow to build a plan modification, negotiate these modifications in multi-robot context and – if the resulting plan is satisfactory – to actually change the executable plan. In that context, the decision control component plays a central role as it is the middle-man between the plan changes brought by the execution and the changes which are being added to transactions.

The plan modification operators described in this chapter show how it is possible, by using basic relation modifications, to build more complex adaptation operators. The `replace_plan` operator uses the notion of fullfilled model to determine if one task can be replaced by another, and if it is the case, allows the replacement. If the replacement involves bringing the new subplan in a given execution state, the use of *transitions* allows to handle non-instantaneous replacements as well. Finally, the `split` operator allows to modify the plan so that it represents an interruption and resume of parts of its current activities.

5 Implementation and results

5.1 Implementation: the Roby application framework

The current implementation of our plan management system is written in Ruby[1], which is an interpreted object-oriented language. As we will see, we chose this language for its expressiveness and flexibility. This last characteristic allowed to develop a very extensible framework, allowing to quickly prototype new features in the system. Finally, the use of an all-purpose language like Ruby allowed to develop an application framework, in which multiple aspects of robotic software development are integrated around the plan manager: testing, simulation, logging, ...

This section presents different aspects of this implementation. First, how task and event models are defined in the Roby system. Second, the bindings between our Roby system and the GenoM[30] functional layer – used on the Dala rover – is presented as an example of how a functional layer can be plugged in our plan management framework. The testing part of the Roby application framework is then outlined. Finally, we present some performance results for the existing implementation.

5.1.1 Definition of tasks and events

Our representation of the relationships between task models is very close to the object-oriented (OO) paradigm: a base model is a more generic model than a model which is built upon it. In OO, a *model* is called a *class*, and the *class inheritance* mechanisms allow to represent the relationships between the different models. Moreover, by using object orientation as a basis for our implementation of models and task instances, we benefit from the reuse of task-related code (event commands, handlers, exception handlers).

[1] `www.ruby-lang.org`

```
class MoveTo < Roby::Task
  abstract
  terminates

  event :start, :controlable => true
  event :blocked
  forward :blocked => :failed
end
```

Figure 5.1: Code example: definition of the generic `MoveTo` task model mentioned on Fig. 2.3

Thanks to Ruby capabilities, we have been able to define a task and event definition language directly in the Ruby language [2]: the model definition code is itself a Ruby script. For instance, to define the abstract `MoveTo` model defined in chapter 2.1, one would write the code on Fig. 5.1.

This code does the following:

- it creates a MoveTo class derived from the generic Roby::Task model. Among other things, it inherits the events already defined in this generic model ($e_{aborted}$ and $e_{success}$ do not have to be redefined) and the relations between these events (for instance, there is no need to read a forwarding relation between $e_{success}$ and e_{stop}).
- it declares that this model is abstract.
- it declares that task instances of this model terminates naturally. This statement is equivalent to writing

  ```
  event :failed do
    emit :failed
  end
  event :stop do
    failed!
  end
  ```

  I.e. to make e_{failed} a pass-through event, and to define a command for e_{stop} which calls the e_{failed} event. This is actually how the terminates statement is defined.
- it creates a new contingent event $e_{blocked}$ (events are contingent by default) and forwards this event to e_{failed}.

Since this model definition language is itself Ruby code, extending the language with new often-used statements like terminates is easy to do. Moreover, it is possible to define classes on the fly, which is used to map *anonymous* local classes to represent unknown classes of remote plan managers.

5.1.2 Binding GenoM into Roby

The functional layer of our Dala rover is made from a set of GenoM functional modules. To represent these modules into our Roby applications, we wrote a small plugin which loads the

[2] these days, this is called an *embedded Domain Specific Language (DSL)*

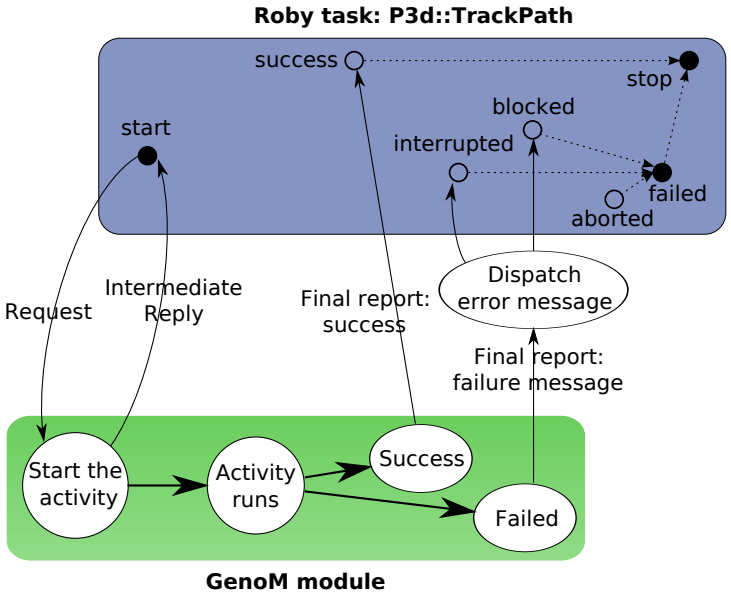


Figure 5.2: How GenoM activities are represented by Roby tasks. The `P3d::TaskPath` defines a `BLOCKED` failure message which, when returned, causes $e_{blocked}$ to be emitted

GenoM module definition files and automatically generates the required models. As defining models is done through Ruby code, this plugin has been quite straightforward to write, and does not require any code generation. This section presents this Roby/GenoM binding: how GenoM modules are defined and how they are mapped onto Roby models.

A given GenoM module defines a set of *requests*, each request corresponding to a possible service which can be performed by the module. Requests accept a set of input parameters and can return a set of output values.

When a request is sent and accepted by the module, the module registers an *activity* – a representation of the request as it is being executed. During the activity lifetime, the module notifies the caller of two things:

- an *intermediate reply* is sent to acknowledge that the module has started to handle the request.
- a *final reply* is sent when the activity has finished. A final reply can either notify of a success or return an error value which represents the failure of the activity. The set of possible errors is specified by the request definition.

When a module is to be used in Roby, the Roby/GenoM bindings load the module definition file and defines a task model for each request available in the module. Each task model has for argument set the set of arguments the request requires. The generic representation of GenoM request by Roby tasks is illustrated on Fig. 5.2.

The GenoM module itself is represented in our framework: a `Roby::Genom::RunnerTask` instance represents, in our plan, the process of the GenoM module itself. Moreover, all requests are **executed_by** this task (see Fig. 2.6). This allows to gracefully handle both an unexpected termination of a GenoM module and to make Roby both start and terminate the process at

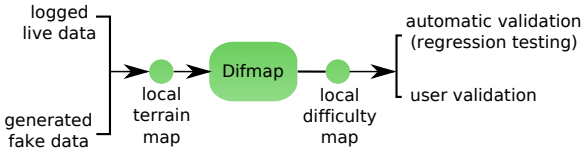


Figure 5.3: Unit-testing a functional module

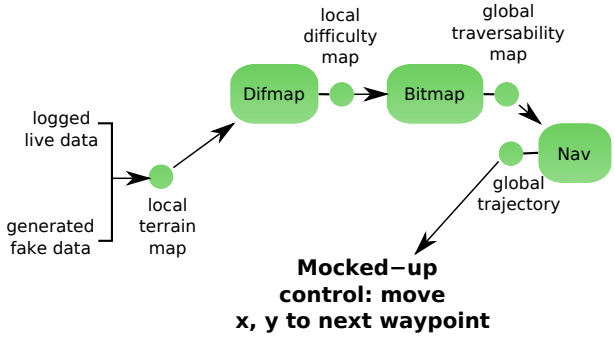


Figure 5.4: Testing the plan generation capabilities of our rover

application start and end. This feature is essential for automatic module testing, which we will discuss in the next section.

5.1.3 Testing

Integration of a robotic application requires *testing*. Because our system is able to manage every components in the system – from the functional layer processes up to the planners – and thanks to the complete integration of the software framework into a generic language it has been possible to integrate a testing framework in the Roby application framework.

Testing a robotic systems can be done at various levels of details:

functional layer service test one service in the functional layer. This involves checking data processing functions against reference dataset or some other simple test datasets and checking the result.

functional layer integration it is the same that the previous, but instead of testing each module separately, we inject datasets at the beginning of the dataflow and check the output at the end of the module chain. This checks that our modules are compatible in a more dynamic way.

plan generation this is much more high-level: the low-level control tasks are replaced by mockups written directly in Ruby. For instance, the path planning algorithm and the plan generated for our Nav::MoveTo task can be tested by using a fake `TrackPath` task. The path tracking task which in the real system is implemented by one of our two motion modalities is replaced by a task which changes the position of the robot linearly towards its goal (Fig. 5.4).

integrated simulation run the whole system in a simulation environment. This requires that

most of the functional layer *can* run in the simulation environment. It is the case for our system [42].

field testing of course.

In the domain of functional layer testing, a common method is to rely on a module service to load some *a priori* data. For instance, our `P3D` module requires the `DTM` module output. One could insert a request in `DTM` to load *a priori* data and make it export that data as if it were the perceived one. This is a mistake for several reasons:

- it makes the testing of the P3D module rely on the internals of the DTM module, which should be avoided during unit-testing.
- the data exported by the source module will often not be incremental. For instance, in the case of the DTM module, one would not dump the internal data everytime, but only once in a while: exporting it continuously would greatly impact the performance of the system. One is therefore not testing the algorithms on the data they will have to manage in the real world.

For those reasons, we developped a tool which is able to log the dataflow exported by our GenoM modules and replay it. These samples are then used during the testing as input of the tested modules.

5.1.4 Performance

On our Dala rover, the average data processing length is 10ms for a mean of 55 tasks in the plan. This is no surprise since all the algorithms used are at most O(N). However, there are performance issues with the Ruby interpreter itself, this is why we fixed the cycle length at 50ms.

The Ruby language is garbage collected. As such, an internal garbage collection process runs whenever needed (Fig 5.5). This is a known issue in realtime applications: the garbage collection process of programming languages is often a performance hog. However, we don't consider it to be a problem in our case as the realtime part of the robot behaviours is handled by the functional layer: a bound of 300ms for the system response time is enough for our applications. If more were needed, time-bounded garbage collection has seen interesting progresses, allowing to have real-time garbage collected languages in embedded systems. It is therefore possible to imagine having a dynamic language like Ruby use a time-bounded garbage collection in the future.

5.2 The Dala/Ressac Experiment

5.2.1 Supervision of the Dala Rover

The supervision of the Dala rover has been the first interesting application for our plan manager. The way the functional layer is designed, the supervision system is supposed to handle the following:

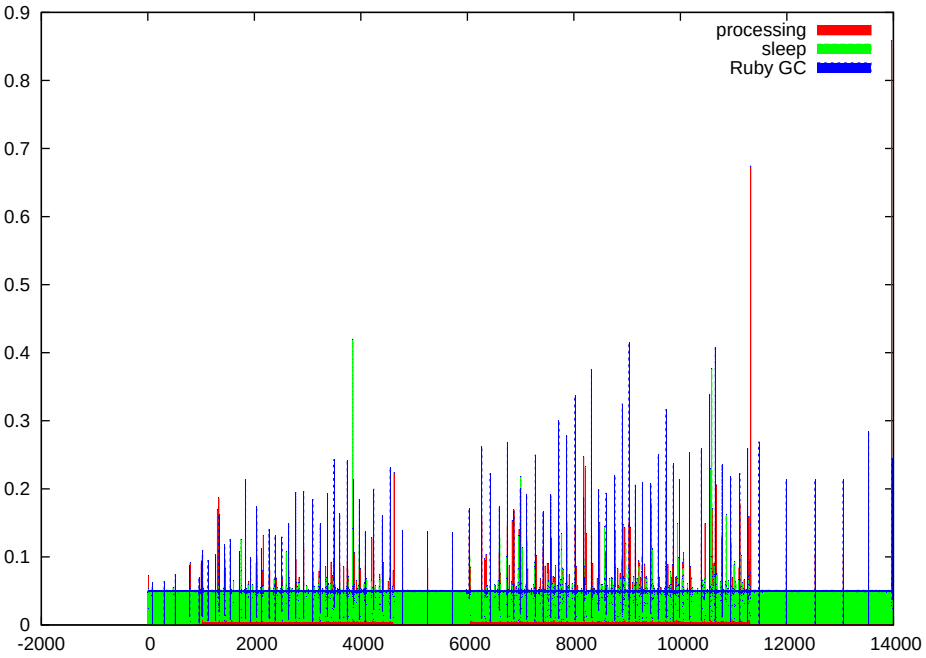


Figure 5.5: Performance measurement of the Roby executive: milliseconds versus cycle index. Since we have a fixed-length cycle, the executive has to sleep to wait the beginning of the next cycle (green). The blue part is the time spent by the Ruby interpreter for its own garbage collection. Aberrant sleep and processing times are certainly due to the lack of a realtime OS on our robot, since the measurement uses `gettimeofday`.

```
event = State.on_delta(:d => 0.5).
    or(:t => 5).
    or(:yaw => 0.3)

event.signal mapping.event(:loop_start)
```

Figure 5.6: Trigger of the DEM mapping loop based on a state event. The e_{loop_start} event of the mapping task is called if the robot moves for more than 30cm, terns for more than 0.3rad or more than 5 seconds have passed.

- the digital terrain map and the traversability map updates are not automatic. The supervision system is supposed to trigger them when needed.
- due to limitations of the perception system, specific error handling procedures have been defined for the integration of the Nav/P3D couple.

The perception loops has used the implementation of the following constructs:

state events which are emitted when a pre-defined state is reached, or when the state has changed more than a given threshold. This events are used in our case to trigger the perception based on threshold on time, position and heading (Fig. 5.6).

loop construct the `PlanningLoop` task is a planning task which generates the same procedure in loops (Fig. 5.7). This generation can be either triggered periodically or by the command of the e_{loop_start} event. Since our goal is to represent the future of the system, it is possible to manage the loop ahead of time: to develop a fixed amount of iterations before they are needed.

The P3D motion modality uses a digital elevation map (DEM) generated from stereovision. Defaults in the DEM are quite common since the errors in localization – in particular attitude estimation – have big effects on the resulting DEM. To mitigate those effects, a two-stage error handler is installed on the `P3d::TrackPath` task. This error uses the **error_handling** relation so that it is transparent to any other task using the P3D motion. This scheme makes the P3D implementation more robust to perception problems, without the rest of the system noticing.

1. the first handler only does a perception of the front of the robot.
2. if the movement still fails after this DEM update, the DEM is completely reinitialized and we do a map of the robot surroundings through a sequence of perceptions around the robot.
3. if the movement still fails, then the task is not repaired: an exception is raised.

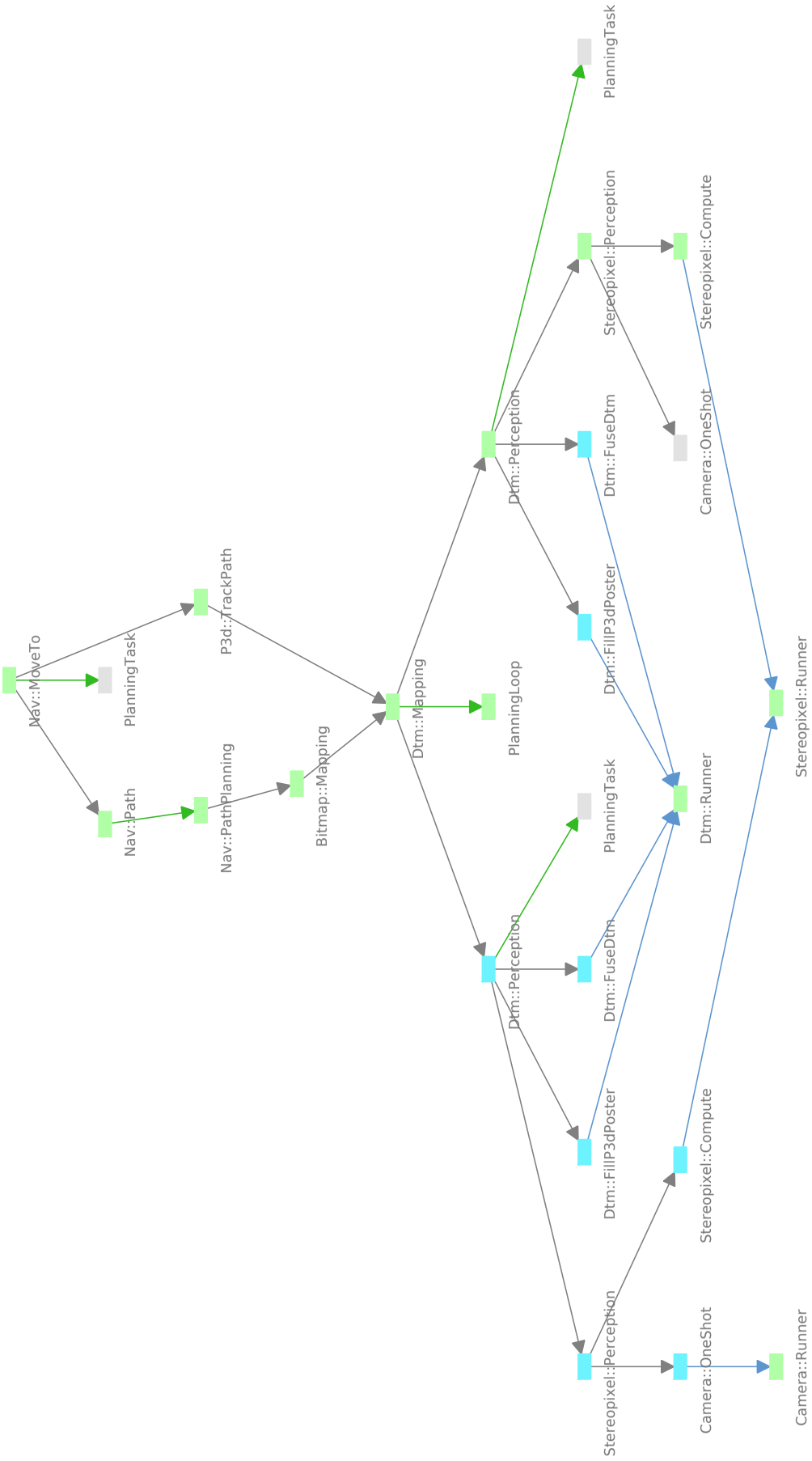


Figure 5.7: Representation of the Dtm perception loop in the real robot. Two patterns are developped at this point: one pattern is running (on the right) and one has been developped ahead of time.

In this latter case, the robot is most likely in a situation where the long range path planner, nav, and the local motion modality do not agree on their model of the environment: P3D cannot reach the target given by nav. Because of that, if the `P3d::TrackPath` fails, an exception handler on `Nav::MoveTo` updates Nav's model of the environment to mark the zone in front of the robot as not traversable. The P3D motion task is then restarted. This exception handler is independent of the actual motion modality used: it is a repair which is defined by the parent of a **depends_on** relation to mitigate problems in its interactions with its child.

In the three cases (two local repairs using the **error_handling** relation and one using an exception handler), the repair schemes are reset when the robots moves significantly (in our case, the threshold is set at 1 meter). For instance, when this treshold is met, the current **error_handling** relation put on the `P3d::TrackPath` task is replaced by the subplan for the first stage handler.

5.2.2 Cooperation: simulation results

The cooperation has been tested in simulation. For this simulation, few modules have been modified to read the robot state from the simulation system instead than from the hardware – while keeping the same interfaces – and two modules have been removed (Fig. 5.8). This allows to have very few differences between the controller used in simulation and the one used on the real robot.

The execution of the cooperation is a cycle: the rover generates regions of interest based on the traversability map, the UAV perceives these regions and updates the rover map (Fig. 5.9). In our current implementation of the scenario, the UAV is able to perceive at two altitudes. This information is present in the regions generated by the rover: it chooses a preferred zone size based on the information which is currently available: if very few information is available around the planned path, then a rough high-altitude zone is inserted in order to quickly acquire traversability information. If the zone is already known, a low-altitude perception is inserted to refine the knowledge of that particular area.

The initial terrain and the apriori traversability maps used to simulate the UAV perception are shown on Fig. 5.10. The UAV maps are generated based on the elevation data based on limits on the terrain slope and random confidence. For the high altitude map, the confidence is within the $[0.12, 0.25]$ range, while it is within the $[0.25, 0.5]$ range for low altitude perception. Images of the navigation progressions are shown on Fig. 5.11. The whole navigation is done in roughly 30 minutes for 400 meters at maximum 1 m/s for the rover.

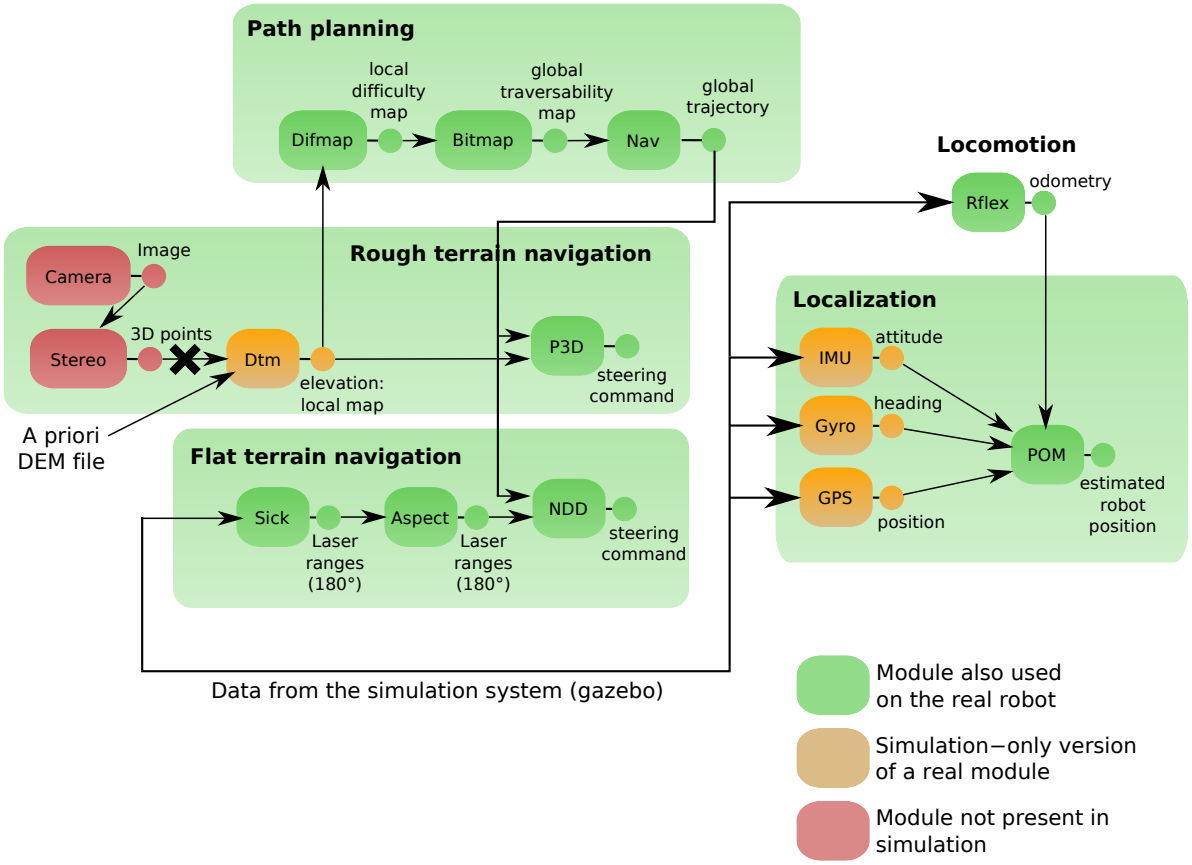
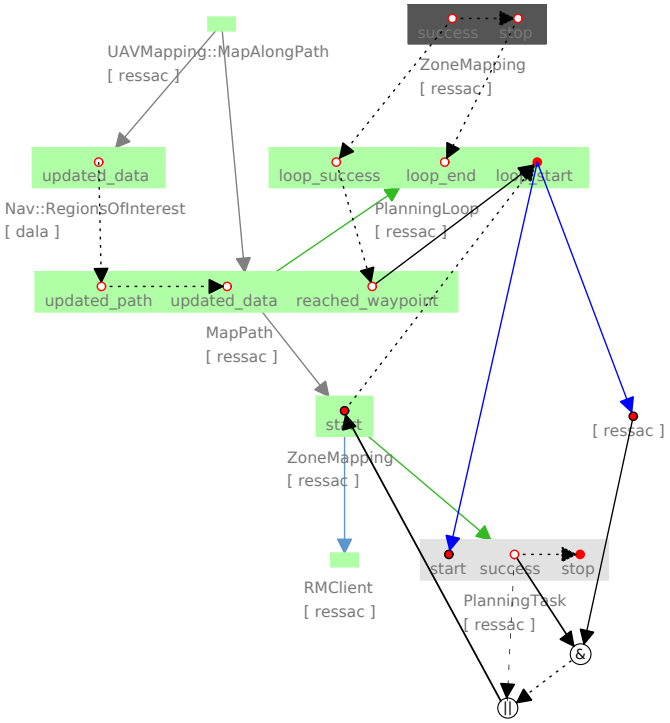
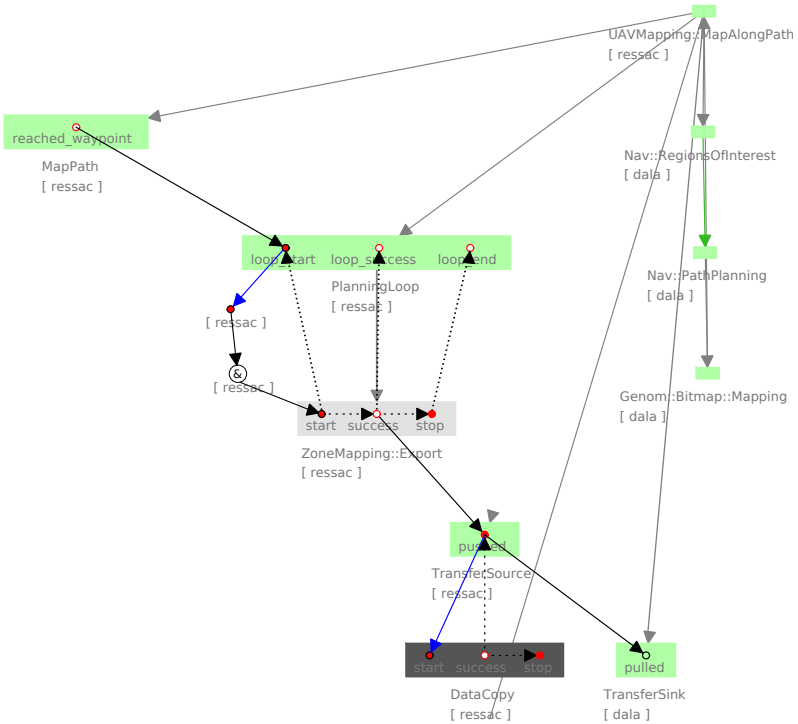


Figure 5.8: GenoM functional layer of the Dala rover used in simulation. Modules like Rflex are interfaced with a simulation-only version of their hardware access library: the servoing or data processing code remains untouched.

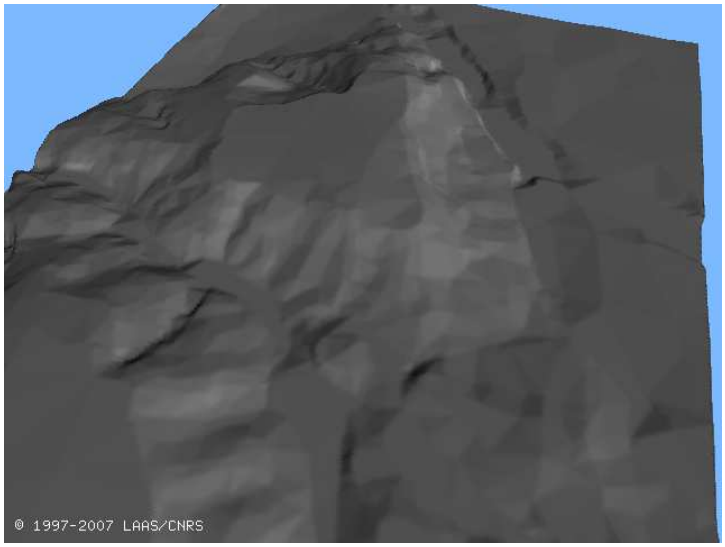


(a) The rover generates a path and a list of regions of interest. The UAV is notified of this update through a $e_{updated\_data}$ event. The path itself is also transmitted by Roby, as the internal data of the `RegionsOfInterest` task. The UAV then starts mapping based on the information contained in the `RegionsOfInterest` task. It chooses a region to perceive and starts the mapping (`ZoneMapping` task). The Roby controller cannot, at this point, interrupt a `ZoneMapping` task. Therefore, the loop simply waits for the end of the previous ZoneMapping task before starting a new one. The event structure at the bottom-right is part of the internal implementation of the `PlanningLoop task`

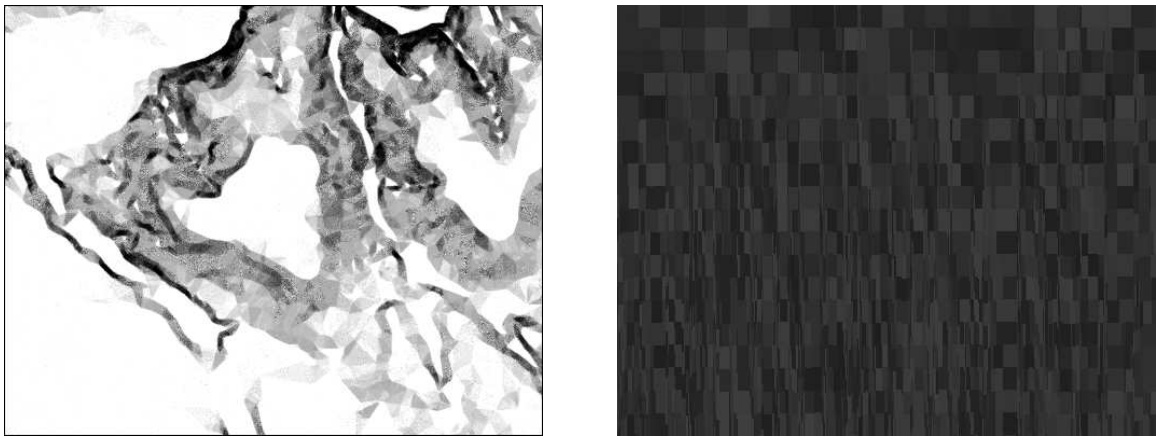


(b) When the mapping is finished, the rover is notified that new information is available. The data itself is passed through another channel established by the `TransferSink/TransferSource` pair. The rover integrates this new information and may update the list of regions of interest. We do not use a joint task for data transfer because the handling of joint tasks is still too experimental in our implementation.

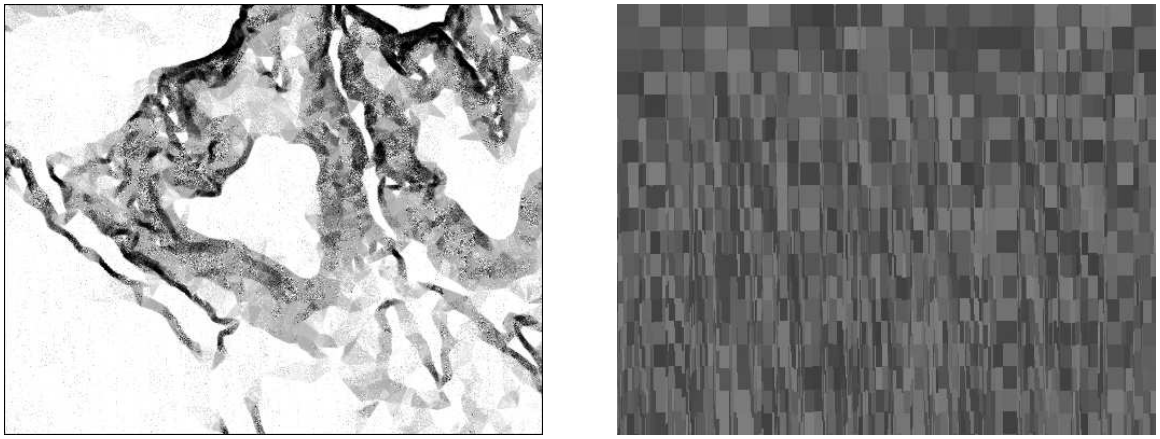
Figure 5.9: Two stages of execution in the rover/UAV cooperation



(a) Elevation map of the terrain used in simulation



(b) Traversability and confidence maps for simulated high altitude perception



(c) Traversability and confidence maps for simulated low altitude perception

Figure 5.10: Maps used for the simulation of our rover/UAV scenario. The confidence maps are random maps generated so that they form small "patches" of constant confidence.

(a) The UAV perceived the first zone for the rover. Low-altitude goals (blue circles) are generated in this newly perceived zone. High altitude goals remain for the terrain still unknown

(b) The rover does not have any perception goal left for the UAV

(c) New perception goals are injected by the rover, to cover an unknown area which may be of interest

(d) Final situation

Figure 5.11: Progression of the UAV/rover cooperation in simulation. The small red/green spheres are the navigation waypoints of the rover, while the blue circles are the regions of interest. The traversability information is displayed in green for fully traversable and red for fully obstacle. Its transparency is proportional to the confidence we have in the information: thus, brighter areas are high-confidence data and darker ones low-confidence.

5.3 From the experiment, back to the implementation

The implementation of both controllers and the implementation of the cooperation scenario allowed us to test the basic concepts of the controller.

Transactions All plan generation in the Dala rover is done asynchronously, so as to test the concept of transactions. They have been needed so as to deal properly with the interaction between asynchronous plan building and garbage collection and – when unexpected problems show up – with the reconfiguration of the plan. The use of distributed transactions, queries and triggers for the rover/UAV cooperation have been an elegant way to implement the interaction between our two robots.

Use of a central plan management component The management of all the robot activities in a single system has also been a great asset for the development of our controllers: the system was able to represent and handle the problems which showed up because of bugs in the functional layer or in the controller code being developed.

Extensibility Building more complex objects – as for instance the state events or the planning loops – on top of the basic system presented in this thesis has been quite easy. This shows that our plan model and execution schemes are expressive enough to build complex plans in our system. It also shows that our implementation allows to easily implement these extensions.

6 Conclusion

6.1 Summary

The plan management component presented in this thesis has been designed for the plan-based control of robots – and of robot teams, meeting the requirements for the integration of multiple decision-making tools in a central component:

- a plan model which is both rich enough to represent the information needed for supervision and simple enough that it is possible to translate plans produced through other plan models into it. This plan model has two main contributions: the separation of activity structure and execution flow through the separation of tasks and event relations, and the integration of multi-robot requirements in a rich task-based system.
- an execution scheme for this plan model, which is multi-robot aware. The main contribution is here to define execution mechanisms based on the task/event dichotomy and on the presence of task graphs – as opposed to task trees. Moreover, the execution engine defines multiple ways to define and handle errors, including the capability of representing error handling in the plan itself, and to represent asynchronous plan repairs – in which the system represents the task which is being repaired, the context in which this task was used and the subplan which is repairing it.
- generic tools for plan adaptation. Our main contribution is here the definition of *transactions*: a generic mechanism allowing to safely modify a plan while it is being executed. Transactions are also used in multi-robot contexts to build joint plans, using the transaction as a basis for negotiation. We also developped plan adaptation operators which take explicitely into account the problems related to changing plans while they are being executed, showing that more complex plan adaptation schemes can be built on the plan

model we defined.

The final aim of this software system is both the integration of existing decision-making tools (planning systems, cooperation protocols, ...) and the development of new ones, designed with the specificities of plan execution in mind. This software system has been successfully used in the supervision of a rover/UAV team, for the realization of a cooperation scenario in which the UAV provides information for the rover's navigation. This scenario has allowed to develop two controllers on our software implementation, and to illustrate the mechanisms we described in this thesis.

6.2 Future Work

6.2.1 Extensions to the Roby software system

6.2.1.1 Plan model and plan execution

Explicit representation of time This is the most obvious extension to our plan management. The management of a robotic system cannot be completely done without an explicit representation and management of time. This extension is twofold. First, a time prediction scheme can be implemented through the use of the event networks and duration estimation functions provided by the task models. Second, time constraint networks could be integrated, allowing for instance to express constraints over the instant of emission of events, or how long a **depends_on** relation can remain broken during a plan repair. The integration of time during the execution would indeed require to extend our execution scheme as well.

Scheduling Based on the explicit representation of time outlined above, our system would greatly gain from the integration of a scheduler: since we represent all activities in the system, the scheduling of these activities – and particulary the interactions between planning and execution – would be a great contribution.

State prediction Using our plan model and state prediction functions provided by the task models, we could estimate the future state of the robot based on its current state and the plan itself. Monte-carlo techniques have already been used for that same goal, for instance in Structured Reactive Controllers. A challenge would be a joint prediction of both state and time.

Plan merging In its current state, the plan model only provides the means to determine if two tasks are actually performing the same action. To implement plan merging – removing redundancies in plans – more information would be added to both the plan model and the execution state of the tasks:

- a temporal notion of *effects* for tasks: an event which determines for how long the result of a given task remains reusable by the rest of the plan.
- a notion of *commutativity*: whether or not the execution of the sequence $T_1 + T_2$ is the same than executing the sequence $T_2 + T_1$.

- extending the plan model towards multi-robot constraints: specifying the set of agents which could execute the child of a relation, allowing to know if it is possible to replace the task of one agent by the task of another one.

A first step towards the implementation of plan merging is the online removal of redundancies: whenever a task is started, check if another task is not realizing the same goal. A more interesting scheme is to *plan* the merge: determine ahead of time the tasks which are most likely to be merged and modify the plan accordingly.

6.2.1.2 Plan management

While we believe that the current system can integrate multiple planning tools, it is not done in practice – apart from a prototype translation plugin for IxTeT. This area must definitely be extended. State and time prediction in plans would be an asset here: it would allow us to create planning problems based on the future estimated state of the robot, instead of building the plan from its current state.

6.2.2 Perspectives

The capabilities of robotic systems would greatly gain by focusing on the integration of the various subsystems needed to make a robot work. While the integration of functional layer has already seen extensive work, not a lot is done towards the integration of decision-making components. From our point of view, the integration of fault-tolerance and diagnosis at the plan level could greatly benefit the overall robot reliability. Moreover, if possible, the integration of such tools in a plan manager like our own would allow a greater reusability of the separate tools.

To achieve that, the plan model we described in this thesis, and the execution scheme built upon it, could gain by a formal definition. Such a formal definition could for instance allow to prove the equivalence between plan models.

Bibliography

[**1**] R. Alami and S. Botelho. Plan-based multi-robot cooperation. In *Advances in Plan-Based Control of Robotic Agents*, Lecture Notes in Computer Science. Springer, 2001.

[**2**] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17(4):315–337, apr 1998.

[**3**] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and S. Qutub. Operating a large fleet of mobile robots using the plan-merging paradigm. In *Proceedings of the IEEE ICRA*. IEEE, 1997.

[**4**] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and F. Robert. Multi-robot cooperation in the martha project. *IEEE Robotics and Automation Magazine*, 1998.

[**5**] R. Alami, F. Ingrand, and S. Qutub. A scheme for coordinating multi-robot planning activities and plans execution. In *Proceedings of European Conference On Artifical Intelligence*, 1998.

[**6**] R. Arkin, T. Collins, and Y. Endo. Tactical mobile robot mission specification and execution. In *Proceedings of Mobile Robots XIV*, 1999.

[**7**] R. C. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *Proceedings of the International Conference On Robotics and Automation ICRA'87*, 1987.

[**8**] R. C. Arkin and T. R. Balch. Aura: Principles and practice in review. *Journal of Experimental and Theoretical Artificial Intelligence(JETAI)*, pages 175–188, 1997.

[**9**] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: A taxonomy. In *Building the information society, 18th IFIP World Computer Congress*, 2004.

[**10**] C. Barrouil and J. Lemaire. Advanced real-time mission management for an AUV. In *SCI NATO RESTRICTED Symposium on Advanced Mission Management and System Integration Technologies for Improved Tactical Operations*, 1999.

[**11**] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods*, 2006.

[**12**] M. Beetz. *Concurrent Reactive Plans*. Springer-Verlag, 2000.

[**13**] M. Beetz, T. Arbuckle, T.Belker, A. B. Cremers, D. Schultz, M. Bennewitz, W. Burgard, D. Hähnel, D. Fox, and H. Grosskreutz. Integrated, plan-based control of autonomous robots in human environments. *IEEE Intelligent Systems*, 2001.

[**14**] M. Beetz and H. Grosskreutz. Causal models of mobile service robot behavior. In *Proceedings of AIPS*, pages 163–170, 1998.

[**15**] DE. Bernard, GA. Dorais, C. Fry, EB. Gamble Jr., B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, P. Pandurang Nayak, B. Pell, K. Rajan, N. Rouquette, B. Smith, and BC. Williams. Design of the remote agent experiment for spacecraft autonomy. In *Proceedings of the IEEE Aerospace Conference*, 1998.

[**16**] R. P. Bonasso, D. Kortenkamp, and T. Whitney. Using a robot control architecture to automate space shuttle operations. In *Proceedings of the 1997 National Conference on Artificial Intelligence*. AAAI, 1997.

[**17**] D. Bonnafous, S. Lacroix, and T. Siméon. Motion generation for a rover on rough terrains. In *International Conference on Intelligent Robots and Systems, Maui, Hawai (USA)*, Oct. 2001.

[**18**] S. C. Botelho and R. Alami. M+: a scheme for multi-robot cooperation through negotiated task allocation and achievement. In *Proceedings of IEEE ICRA*, 1999.

[**19**] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[**20**] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proceedings of AIPS*, 2004.

[**21**] C. Chouinard, T. Estlin, D. Gaines, and F. Fisher. Intelligent rover decision-making in response to exogenous events. In *Proceedings of i-SAIRAS*, 2005.

[**22**] P. Cohen, H. Levesque, and I. Smith. On team formation. *Contemporary Action Theory*, 1998.

[**23**] K. Decker and V. Lesser. Generalizing the Partial Global Planning Algorithm. *International Journal on Intelligent Cooperative Information Systems*, 1(2), 1992.

[**24**] Keith Decker. TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. In *Foundations of Distributed Artificial Intelligence, Chapter 16*, pages 429–448. G. O'Hare and N. Jennings (eds.), Wiley Inter-Science, January 1996.

[**25**] B. Dias. *TraderBots: A New Paradigm For Robust and Efficient Multirobot Coordination in Dynamic Environments*. PhD thesis, The Robotics Institute - Carnegie Mellon University, 2004.

[**26**] E.H. Durfee and V.R. Lesser. Partial Global Planning: A Coordination Framework for Distributed Hypothesis Formation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(5):1167–1183, September 1991.

[**27**] T. Estlin, R. Volpe, I. A. D. Nesnas, D. Mutz, F. Fisher, B. Engelhardt, and S. Chien. Decision-making in a robotic architecture for autonomy. In *Proceedings of 6th i-SAIRAS*, 2001.

[**28**] Patrick Fabiani, Vincent Fuertes, Guy Le Besnerais, Alain Piquereau, Roger Mampey, and Florent Teichteil. ReSSAC: UAV exploring, deciding and landing in a partially known environment. In *Proceedings of the IFAC IAV Symposium*, 2007.

[**29**] A. Finzi, F. Ingrand, and N. Muscettola. Robot action planning and execution control. In *Proceedings of IWPSS*, 2004.

[**30**] S. Fleury, M. Herrb, and R. Chatila. Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture. In *Proceedings of IROS*, 1997.

[**31**] J. Gancet. *Systemes multi-robots aeriens : architecture pour la planification, la supervision et la cooperation*. PhD thesis, Institut Polytechnique de Toulouse, 2005.

[**32**] J. Gancet, G. Hattenberger, R. Alami, and S. Lacroix. Task planning and control for a multi-UAV system: architecture and algorithms. In *Proceedings of IEEE IROS*, 2005.

[**33**] J. Gancet and S. Lacroix. Pg2p: A perception-guided path planning approach for long range autonomous navigation in unknown natural environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, Las Vegas (USA)*, Oct. 2003.

[**34**] E. Gat. ESL: a language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the IEEE Aerospace Conference*, 1997.

[**35**] E. Gat. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*. AAAI Press, 1998.

[**36**] B. P. Gerkey and M. J. Mataric. Sold!: Auction methods for multirobot coordination. *IEEE Transactions on Robotics and Automation*, 2002.

[**37**] B. P. Gerkey and M. J. Mataric. Multi-robot task allocation: analyzing the complexity and optimality of key architectures. In *Proceedings of IEEE ICRA*, 2003.

[**38**] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[**39**] F. Gravot, S. Cambon, and R. Alami. aSyMov: a planner that deals with intricate symbolic and geometric problems. In *11th International Symposium on Robotics Research*, 2003. Invited paper.

[**40**] F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1996.

[**41**] A. K. Jonsson, P. H. Morrisa, N. Muscettola, K. Rajan, and B. D. Smith. Planning in interplanetary space: Theory and practice. In *Proceedings of Artificial Intelligence Planning Systems AIPS'00*, pages 177–186, 2000.

[**42**] Sylvain Joyeux, Alexandre Lampe, Simon Lacroix, and Rachid Alami. Simulation in the LAAS architecture. In *Workshop in Software development in robotics, ICRA 2005*, 2005.

[**43**] S. Lemai-Chenevier. *IxTeT-eXeC: Planning, Plan Repair and Execution Control with Time and Rescource Constraints*. PhD thesis, Institut Polytechnique de Toulouse, 2004.

[**44**] T. Lemaire, R. Alami, and S. Lacroix. A distributed tasks allocation scheme in multi-UAV context. In *IEEE 2004 International Conference On Robotics and Automation (ICRA)*, 2004.

[**45**] 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):87–143, July 2004.

[**46**] B. Lussier, M. Gallien, J. Guiochet, F. Ingrand, M. Killijian, and D. Powell. Planning with diversified models for fault-tolerant robots. In *Proceedings of ICAPS*, 2007.

[**47**] Maja J. Matarić. Behavior-based control: Examples from navigation, learning, and group behavior. *Journal of Experimental & Theoretical Artificial Intelligence*, 1997.

[**48**] J. Minguez, J. Osuna, and L. Montano. A divide and conquer strategy based on situations to achieve reactive collision avoidance in troublesome scenarios. In *Proceedings of IEEE ICRA*, 2004.

[**49**] N. Muscettola, G. A. Dorals, C. Fry, R. Levinson, and C. Plaunt. IDEA: Planning at the core of autonomous reactive agents. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, October 2002.

[**50**] L. Nana, F. Singhoff, J. Legrand, J. Vareille, P. Le Parc, F. Monin, D. Massé, L. Marcé, J. Opderbecke, M. Perrier, and V. Rigaud. Embedded intelligent supervision and piloting for oceanographic AUV. In *Proceedings of the Oceans conference*, 2005.

[**51**] I Nesnas, R Volpe, T Estlin, H Das, R Petras, and D. Mutz. Toward developing reusable software components for robotic applications. In *Proceeding of the International Conference on Intelligent Robots and Systems IROS*, 2001.

[**52**] I.A.D. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I-H. Shu, and D. Apfelbaum. CLARAty: Challenges and steps toward reusable robotic software. *International Journal of Robotic Research*, 3(1):23–30, 2005.

[**53**] L. E. Parker. Alliance: An architecture for fault tolerant, cooperative control of heterogeneous mobile robots. In *Proceedings of IROS*, pages 776–783, 1994.

[**54**] L. E. Parker. Adaptive heterogeneous multi-robot teams. *Neurocomputing, special issue of NEURAP '98: Neural Networks and Their Applications*, 28:75–92, 1999.

[**55**] P. Paruchuri, M. Tambe, F. Ordonez, and S. Kraus. Towards a formalization of teamwork with resource constraints. In *Proc. of the International Joint Conference On Autonomous Agents and Multiagent Systems, AAMAS 2004*, 2004.

[**56**] T. Peynot and S. Lacroix. A probabilistic framework to monitor a multi-mode outdoor robot. In *Proceedings of IEEE IROS*, 2005.

[**57**] Louise Pryor and Gregg Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[**58**] F. Py and F. Ingrand. Dependable execution control for autonomous robots. In *Proceedings of IEEE IROS*, 2004.

[**59**] D. V. Pynadath and M. Tambe. Multiagent teamwork: Analyzing the optimality and complexity of key theories and models. In *Proceedings of the International Conference On*

Autonomous Agents and Multiagent Systems. ACM, 2002.

[60] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Educatio, 2003.

[61] N. Schurr, S. Okamoto, R. T. Maheswaran, P. Scerri, and M. Tambe. Evolution of a teamwork model. *Cognitive Modeling and Multi-Agent Interactions*, 2005.

[62] R. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 1994.

[63] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings of IEEE IROS*, 1998.

[64] R. Simmons and E. Coste-Manière. Architecture, the backbone of robotics systems. In *Proceedings of ICRA*, 2000.

[65] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, and J. Sullivan. A layered architecture for office delivery robots. In *First International Conference On Autonomous Agents*, pages 235 – 242, 1997.

[66] R. Simmons, T. Smith, M. B. Dias, D. Goldberg, D. Hershberger, A. Stentz, and R. M. Zlot. A layered architecture for coordination of mobile robots. In *Proceedings From the NRL Workshop On Multi-Robot Systems*. Kluwer Academic Publishers, 2002.

[67] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In *IEEE Transaction on Computers*, number 12 in C-29, pages 1104–1113, 1980.

[68] M. Tambe. Agent architectures for flexible, practical teamwork. In T. Senator and B. Buchanan, editors, *Proceedings of the Fourteenth National Conference On Artificial Intelligence and the Ninth Innovative Applications of Artificial Intelligence Conference*, pages 22–28, Menlo Park, California, 1996. American Association For Artificial Intelligence, AAAI Press.

[69] M. Tambe. Teamwork in real-world, dynamic environments. In Victor Lesser, editor, *Proceedings of the Second International Conference On Multiagent Systems (ICMAS'96)*, Kyoto, Japan, 1996.

[70] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 9 1997.

[**71**] Vandi Verma, Ari Jonsson, Corina Pasareanu, and Michael Iatauro. Universal executive and PLEXIL: Engine and language for robust spacecraft control and operations. In *AIAA Space Conference*, 2006.

[**72**] R Volpe, I Nesnas, T Estlin, D Mutz, R Petras, and H. Das. CLARAty: Coupled layer architecture for robotic autonomy. Technical report, NASA Jet Propulsion Laboratory, 2000.

[**73**] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty architecture for robotic autonomy. In *Aerospace Conference*, pages 121–132, 2001.

[**74**] B. C. Williams, M. Ingham, S. Chung, P. Elliott, and M. Hofbaur. Model-based programming of fault-aware systems. *AI Magazine*, 2004.

[**75**] Q. Yang. *Intelligent planning: a decomposition and abstraction based approach.* Springer, 1997.

Résumé en français

Introduction

Le sujet de cette thèse est le développement d'un composant logiciel de gestion de plan, développement motivé par la nécessité de développer des architectures où les systèmes de prise de décision peuvent être intégrés, et où la séparation entre les différents composants de décisions est éliminée.

Les principales contributions de cette thèse sont :

– la définition et l'implémentation d'un tel composant logiciel, permettant la gestion de plan dans un contexte multi-robot. Ce composant intègre les outils nécessaires à la construction de plans, leur exécution, leur adaptation et la gestion d'erreur durant cette exécution.

– la définition d'une architecture basée sur ce composant, mettant en avant différentes fonctions nécessaires à l'intégration de plusieurs systèmes de décision dans un même robot.

– la mise en adéquation d'une telle architecture à la problématique du multi-robot.

– la mise en place d'une expérimentation de coopération rover/UAV démontrant les capacités du système présenté dans ce manuscrit.

Dans un premier chapitre, nous analysons les forces et les faiblesses des approches décrites dans la littérature, dans les cadres du mono et du multi-robot. Au cours de cette analyse, nous présentons quelles sont, à notre avis, les problèmes qu'une architecture robotique devrait résoudre vis-a-vis des systèmes de décision. Enfin, nous présentons notre approche et un scénario qui servira d'exemple au long de ce manuscrit et qui a servi de support à la validation expérimentale de notre approche.

Au cours du second chapitre, nous présentons ce qu'est un plan et comment il est représenté dans notre système.

Le chapitre 3 décrit comment un plan est exécuté par notre système. Ce chapitre présente également comment les situations non nominales peuvent être gérées.

Le chapitre 4 décrit comment, dans notre architecture, les plans peuvent être modifiés en cours d'exécution.

Au cours du chapitre 5, nous décrivons certains points clefs de notre implémentation. Nous présentons également des résultats obtenus au cours de l'implémentation de notre scénario de coopération rover/UAV.

Enfin, le chapitre 6 résume nos contributions et discute des limitations de notre système et comment nous le voyons évoluer.

1 Problématique

La gestion d'un robot autonome – sans parler d'équipes de robots – est une problématique difficile : il s'agit d'intégrer de nombreuses fonctionnalités pour donner au(x) robot(s) un comportement autonome dans un environnement complexe, partiellement connu, et dynamique. Pour permettre l'intégration de ces fonctionnalités, des *architectures* ont été définies : ce sont des principes et outils guidant l'intégration.

Le sujet principal de cette thèse est l'intégration d'outils de decision. Ce chapitre présente dans un premier temps quelles sont les approches principales en matière d'achitecture pour un robot seul, comment la décision y est traitée et la limitation de ces approches ¡¡classiques¿¿. Puis, nous présentons des architectures plus atypiques qui ont essayé de corriger ces défauts, et enfin les problèmes liés à la gestion d'erreurs. Dans un deuxième temps, nous présentons les approches multi-robot, toujours en centrant notre discours autour de la notion de prise de décision. En se basant sur ces considérations, nous présentons notre approche puis un scénario illustratif qui sera utilisé tout au long de ce manuscrit.

1.1 Prise de décision dans un robot seul

1.1.1 Principales approches dans les architectures mono-robot

En matière d'architectures, deux approches principales se sont imposées ces dix dernières années : la première approche a été fondée sur l'idée que la partie ¡¡décision¿¿ du cycle classique perception-décision-action pouvait être omise : il est possible d'atteindre un comportement autonome sans définir d'outils spécifiques pour la prise de décision. Cette approche est dite ¡¡comportementale¿¿ car elle se base sur l'idée que, à long terme, l'émergence d'un comportement autonome peut être basé purement sur la définition de réactions aux stimuli de l'environnement

et à la présence de stimuli ¡¡motivateurs¿¿.

L'autre approche est fondée sur l'intégration d'outils spécifiques à la prise de décision pour contrôler un système comportemental. L'idée est donc de contrôler un système réactif – la couche comportementale – à partir d'un système capable de raisonner à plus long terme. Ces approches ont pris le nom d'approches ¡¡hybrides¿¿ ou ¡¡trois couches¿¿ car basées sur une couche décisionnelle, une couche comportementale et une couche de traduction entre les deux.

Notre travail s'inscrit dans la approche : à notre avis, comportements et prise de décision à long terme sont complémentaires. La suite de cette thèse présente donc les structures de décision dans les architectures hybrides multi-couches, quels sont les limitations de ces architectures et comment d'autres architectures plus atypiques ont tenté de corriger ces limitations.

1.1.2 Où sont prises les décisions ?

La décision n'est pas un processus localisé : nous pouvons définir un processus de décision comme tout processus qui fait des *choix* : un processus qui limite l'ensemble des futurs possibles du robot. Alors que les processus de planification appartiennent clairement à cette catégories, il est plus difficile de se prononcer sur d'autres processus comme la planification de chemin.

Délimiter les processus de décision est important : par de mauvaises interactions, il est possible qu'une décision soit prise qui limite les options d'autres processus de décision, et donc les possibilités du robot. Il est donc important qu'une architecture permette le dialogue entre les différents processus de décision.

1.1.3 Les effets de la séparation d'information

L'exemple du planificateur de chemin n'a pas été pris par hasard : dans la plupart des cas, un seul chemin est renvoyé par ceux-ci. L'emplacement physique du robot ayant, de toute évidence, une influence énorme sur ses possibilités d'actions, ce processus – pour être optimal – devrait prendre en compte la planification d'action : les actions rendues impossible par la prise de tel ou tel chemin, les possibilités d'interactions, ... Toutefois, une telle approche est trop complexe. Il faut donc définir un entre-deux permettant un dialogue riche entre les différents planificateurs : partager plus d'informations que dans les systèmes hiérarchiques classiques, mais moins que dans un planificateur qui prendrait explicitement en compte tous les paramètres du problème.

1.1.4 Vers des représentations unifiées

De plus, une telle zone d'échange est également rendue nécessaire par le fait que les différents planificateurs manipulent des modèles différents. Cette partie du problème est traitée par certaines architectures apparues récemment, qui présentent une représentation unifiée de la notion de plan et permettent ainsi l'intégration de plans provenant de plusieurs planificateurs, garantissant leur cohérence.

1.1.5 Représenter et gérer les erreurs

La représentation et la gestion d'erreurs peut être séparée en trois parties :

1. prendre les ¡¡bonnes¿¿ décisions : ne jamais conduire le système dans une situation qui n'a pas été prévue. C'est le rôle des systèmes de planification.
2. définir et détecter des erreurs, qu'elles viennent des couches de décision ou de la perception. C'est le rôle du diagnostic.
3. récupérer les erreurs qui ont été détectées mais qui ne sont pas prises en compte par le modèle utilisé par les planificateurs. C'est le rôle des systèmes dit de *supervision* : récupérer des situations non-nominales sans qu'elles aient été prises en compte par les outils de planification.

Le problème majeur entre ces trois points est la capacité à *représenter* les erreurs : la détection d'erreur n'est pas utile si le système de supervision n'est pas capable de la représenter. Un autre problème est d'être capable de représenter la reprise sur erreur : comment – lorsqu'une erreur est détectée – la gérer pour qu'elle ne nuise pas au bon fonctionnement du robot.

1.2 Systèmes multi-robot

1.2.1 Décision dans les systèmes multi-robot

La première différence entre la prise de décision dans un système multi-robot et dans un système mono-robot est qu'en multi-robot la décision doit décider de qui fait quoi. Deux approches principales existent :

– l'allocation de tâche, qui réduit le problème à décider de quel agent doit être responsable de quelle tâche.

– l'allocation de rôles, qui définit la notion d'équipe : il ne s'agit plus d'allouer un agent pour chaque tâche à réaliser, mais de définir, de manière globale, quelle doit être le comportement de chaque agent au cours du temps pour mener à bien la mission de l'équipe.

La deuxième différence est qu'il existe beaucoup plus de possibilités d'interaction entre les actions des différents robots : notion d'opportunisme, conflits entre robots, ...

1.2.2 Exécution de plans multi-robots

Le problème principal pour l'exécution de plans multi-robot est que les différents robots ne peuvent pas communiquer à tout instant. Ce problème commence à être géré en planification par les extensions décentralisées des problèmes de planification probabilistes. Toutefois, la représentation de l'échange d'information dans les systèmes de planification n'est pas encore très développée.

1.3 Notre approche

Notre approche est centrée autour d'un composant de *gestion de plans* qui présente les caractéristiques suivantes :

1. représente toutes les activités des robots, ainsi que leurs interactions.

2. définit un système d'exécution générique pour le plan ainsi définit.
3. fournit les outils nécessaires à sa modification en ligne : modifier et exécuter les plans simultanément.
4. fournit une base de réflexion autour de la notion de prise de décision en ligne : quelles décisions doivent être prises durant l'exécution des plans pour permettre par exemple de gérer des défauts du plan.

1.4 Scénario illustratif

Pour illustrer les capacités de notre approche, nous avons mis en place – à la fois en simulation et sur le terrain – un scénario basé sur la navigation d'un robot terrestre en environnement inconnu. Ce robot terrestre est aidé par un robot aérien : ce dernier peut fournir une information de traversabilité sur le terrain.

Le rôle de ce scénario est double :

– la couche fonctionnelle du robot terrestre, Dala, est assez riche pour illustrer les capacités de notre système en tant que superviseur.

– l'interaction entre Dala et le robot aérien Ressac permet d'illustrer la création et l'exécution de plans multi-robots.

2
Un modèle de plan

Ce chapitre présente comment nos plans permettent de représenter les différentes activités du système, leurs interactions et le flot d'exécution. Nous présentons également comment ce modèle est adapté aux systèmes multi-robot.

2.1 Composition des plans

Les plans sont composés de deux types d'objets : les *tâches* représentent les différentes activités du robot et les *évènements* représentent des instants remarquable durant l'exécution du plan. Cette séparation permet une plus grande expressivité que dans d'autres modèles de plans où les deux notions (flot d'exécution et activités) sont étroitement liées.

2.1.1 Représentation du flot d'exécution : évènements

Les évènements présents dans le plan représentent les instants remarquables de l'exécution. Un exemple d'évènement est ainsi e_{brakes_on} qui est émit (ou atteint) à l'instant où les freins du robot sont mis, ou les évènements e_{start} des activités. Un sous-ensemble de ces évènements sont contrôlables : leur émission peut être provoquée par le système grâce à la présence d'une *commande* qui, dans notre implémentation, est une procédure à exécuter pour s'assurer que l'évènement sera émis.

La évènements sont structurés par deux relations, qui définissent la réaction du système lorsqu'un ou plusieurs évènements sont atteints :

– la relation *signal* $e_1 \xrightarrow{sig} e_2$ déclare que la commande de l'évènement contrôlable e_2 doit être appelée quand l'évènement e_1 est émis.

– la relation *forward* $e_1 \xrightarrow{fwd} e_2$ déclare que l'évènement e_2 doit être émis quand e_1 l'est.

Cette relation définit une notion d'équivalence : e_2 est une généralisation de l'évènement e_1 car e_2 sera toujours émis quand e_1 l'est. Comme nous allons le voir, un exemple d'utilisation de cette notion est la représentation des différents modes de fautes dans les activités.

2.1.2 Représentation des activités : tâches

Dans notre système, les activités sont représentées par des tâches. Ces tâches sont définies par un modèle basé sur trois ensembles :

– un ensemble d'évènements qui représentent les instants remarquables de l'exécution de la tâche. Le modèle générique de tâche définit par exemple les évènements e_{start}, $e_{success}$, e_{failed} et e_{stop}.

– un ensemble de relations internes entre ces évènements. Comme $e_{success}$ et e_{failed} sont des cas particuliers de e_{stop}, deux relations *forward* sont toujours définies.

– un ensemble d'arguments qui permettent de paramétrer une instance de tâche particulière, en se basant sur un modèle générique. Le modèle de mouvement `MoveTo` prend par exemple deux arguments : x et y.

Un ensemble de relations, présentées plus loin, permettent de représenter comment ces activités interagissent entre elles.

2.1.3 Hiérarchie de modèles et principe de substitution

Afin de représenter les relations de généralisation entre modèles, les différents modèles de tâches forment un arbre. Par exemple, le modèle `MoveTo` mentionné plus haut est spécialisé, dans notre robot terrestre, par les modèles de mouvements de deux modalités de déplacement `P3d : :MoveTo` et `NDD : :MoveTo`, et est lui même une spécialisation du modèle générique `Roby : :Task`. Cette représentation de l'abstraction permet d'échanger une tâche pour une autre à partir du moment que ces deux tâches remplissent le même rôle : le système peut, par exemple, déterminer qu'il est possible d'échanger deux modalités de déplacement à l'endroit ou une tâche `MoveTo` est nécessaire, mais refusera de remplacer une modalité de déplacement par une tâche de perception.

2.2 Relations entre tâches

Dans nos plans, les tâches sont insérées dans différents graphes acycliques orientés (DAG) qui représentent leurs interactions. Cette section décrit les différentes relations qui sont pour l'instant définies dans notre système.

2.2.1 Dépendance directe : la relation depends_on

Une tâche T_1 dépend directement d'une tâche T_2 si la réalisation de T_1 nécessite l'exécution ou la réalisation de T_2. La relations **depends_on** est définie par trois paramètres :

– un modèle de tâche et un ensemble d'arguments décrivant le type d'activité requis par T_1.

– un ensemble d'évènements désirés $E_{success}$ qui sont les évènements dont l'émission est nécessaire à la bonne exécution de T_1.
– un ensemble d'évènements non désirés $E_{failure}$ dont T_1 interdit l'émission.

2.2.2 Processus de planification : la relation planned_by

Lorsque le plan réalisant une tâche T a été généré par un processus particulier P, une relation **planned_by** est ajoutée entre ces deux tâches. Ainsi, il est possible de représenter le processus de planification lui-même : l'évènement $e_{success}$ de P représente l'instant auquel T pourra être exécuté.

2.2.3 Supports d'exécution : la relation executed_by

Dans un système complet, il est commun – pour des raisons de modularité – que différentes tâches soient réalisées par différents processus (au sens "sytème d'exploitation" du terme) ou différents matériels. La relation de dépendance entre la tâche T et le processus externe E qui l'exécute est particulière : lorsque E échoue, de toute évidence T échoue également sans possibilité de reprise. La relation **executed_by** représente ce lien particulier.

2.2.4 Influence : la relation influenced_by

Dans une relation T_1**influenced_by**T_2, l'exécution de la tâche T_2 améliore ou au contraire rend plus difficile l'exécution de la tâche parente T_1. Cette relation représente donc une dépendance "douce".

2.2.5 Interprétation de la structure des tâches : requêtes et notifications

La richesse de la structure de tâches permet de reconnaître des configurations par la reconnaissance de motifs dans le plan. Notre gestionnaire de plan définit à cette fin la notion de requête : un language de définition de motif (basé sur les modèles de tâches, les arguments, les relations entre tâches) et un système permettant de récupérer l'ensemble des tâches du système correspondant au motif décrit.

En multi-robot, ces requêtes permettent de définir des *notifications* : un gestionnaire de plan installe une requête sur un autre gestionnaire de plan afin que ce dernier le notifie de l'apparition de tâches correspondant au motif. Par exemple, notre UAV installe une notification sur notre rover pour détecter la présence d'une tâche de cartographie de traversabilité. Cette notification étant, dans notre cas, le déclencheur de la mise en place de l'interaction entre le rover et l'UAV.

2.3 Plans multi-robots

2.3.1 Qu'est-ce qu'un plan multi-robot ?

De notre point de vue, les plans multi-robot définissent une forme de contrat entre les différents robots : les robots négocient lorsqu'ils créent leur plan commun (aussi appelé plan

joint) et, lorsque les différents robots acceptent le résultats, ils acceptent également d'être liés à ce résultat.

Toutefois, un robot autonome n'est pas totalement contraint par un plan joint : il peut, si nécessaire, en partir.

2.3.2 La notion d'appartenance

Afin de représenter les différents robots dans un même plan, notre modèle définit la notion d'appartenance : chaque tâche *appartient* à un ensemble de gestionnaires de plans. Si on considère un robot donné, une tâche peut ainsi être *locale* si elle appartient uniquement à ce robot, distante si elle ne lui appartient pas ou jointe si elle appartient à plusieurs robots dont celui-ci.

Afin de limiter les communications et la complexité de la gestion de plan, un robot donné n'a qu'une vue partielle des plans des autres robots. Cette vue contient bien entendu toutes les tâches qui lui appartiennent. Elle contient également toutes les tâches qui sont directement en interaction avec ses propres tâches : elles ont une influence directe sur son propre plan. Enfin, si nécessaire, il peut souscrire explicitement à des tâches distantes.

2.3.3 Représentation des rôles

Un rôle est une représentation abstraite de l'ensemble de tâches que doit réaliser un agent ou un ensemble d'agents afin de permettre la réalisation d'une but pour l'équipe complète.

Dans notre gestionnaire de plan, les rôles peuvent être représentés de deux manières : soit explicitement en associant un des agents à qui appartient la tâche à un rôle, soit via la structure du plan en associant une requête à un rôle : ainsi, le système adaptera la liste des rôles des robots au fur et à mesure de l'adaptation du plan.

2.4 Résumé

Ce chapitre a présenté le modèle de plan qui est utilisé par notre composant de gestion de plan. Les principales contributions de ce modèle sont la séparation entre structure d'activités et flot d'exécution et l'intégration du multi-robot dans un modèle de plan basé sur la notion de tâche.

3
Exécution des plans

Maintenant que nous avons définit comment notre système décrit son plan, nous allons présenter comment ce plan est géré au cours de l'exécution. Nous définissons dans ce chapitre un cycle d'exécution, qui est un cycle de durée fixe, basé sur trois étapes :

1. le système détermine quels évènements doivent être émis pour des raisons extérieures (stimuli). Ces évènements sont propagés sur d'autres évènements via les relations *signal* et *forward*.
2. les erreurs sont détectées et gérées.
3. les tâches qui sont inutiles pour les missions du robot sont automatiquement interrompues, ainsi que les tâches pour lesquelles des erreurs ont été détectées et pour lesquelles ces erreurs n'ont pas été réparées.

3.1 Réaction aux évènements

3.1.1 Propagation locale

Lorsque plusieurs évènements sont émis, d'autres évènements doivent être émis ou appelés, suivant ce qui est définit par les relations *signal* et *forward*. Les différents cas de propagation locale sont représentés sur la Fig. 3.1.

3.1.2 Algorithme de propagation globale

Lors de la propagation, il est nécessaire de respecter un ordre global : l'évènement e_{stop} d'une tâche doit, par exemple, être le dernier évènement de cette tâche à être émis. Afin de respecter ces contraintes d'ordre, nous avons mis en place un algorithme de propagation globale, basé sur

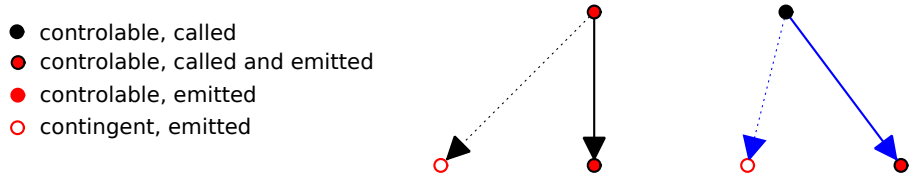


FIG. 3.1 – Propagation locale. A gauche : les différentes représentation d'un évènement durant l'exécution. De gauche à droite : l'émission d'un évènement provoque l'émission ou l'appel d'un autre évènement. Un évènement émet ou appelle d'autres évènements dans sa commande (l'évènement source à droite est seulement appelé, pas émis).

un graphe d'ordonnancement des évènements qui est un sur-ensemble des relations **signal** et **forward**.

3.2 Gestion des erreurs

3.2.1 Définition des erreurs

Notre système définit deux types d'erreurs :

– les erreurs liées au code. Cela peut être le code de notre cadre applicatif lui-même ou le code propre au robot. Ces dernières sont détectées durant la phase de propagation d'évènements.

– les erreurs liées à des violations de contraintes : les différentes relations entre tâches définissent un ensemble de contraintes sur ce qui est nominal pour le plan. Ainsi, une relation **depends_on** décrit un ensemble d'évènements qui ne sont pas acceptables dans le cadre de cette relation. Une relation **planned_by** échoue si la tâche planifiée est abstraite (ne peut pas être exécutée par le système) et si la tâche de planification a échoué. Ces erreurs sont listées dans la deuxième phase du cycle d'exécution en appelant des routines d'analyse globale du plan.

Pour pouvoir être gérée, une erreur particulière doit être associée à un *point d'échec* dans le plan. Ce point d'échec est l'objet du plan (tâche ou évènement) qui a été déterminé comme étant l'objet à la source de l'erreur. Il est nécessaire aux mécanismes de reprises que nous décrivons ici : pour cette raison, les erreurs liées au code de notre applicatif ne peuvent en général pas être reprises. En effet, un problème dans le code de propagation d'évènements, par exemple, ne peut être directement associé à un objet du plan.

3.2.2 Gérer les erreurs

Notre système définit trois manières de reprendre une erreur :

– il est possible de réparer directement durant la phase de propagation. Cela permet de corriger des erreurs dont la reprise est connue et simples sans recourir à des mécanismes plus complexes.

– il est possible de représenter des réparations directement dans le plan : on associe une tâche à un évènement. Quand une erreur qui provient de cet évènement est détectée, la tâche est lancée. Pendant que la tâche de réparation s'exécute, l'erreur est *en cours de réparation* et

le système l'ignore. Lorsque la tâche finit – ou si la réparation prend trop de temps – le plan doit soit être réparé, soit l'erreur est à nouveau prise en compte normalement. La relation de tâches **error_handling** automatise cette procédure.

– l'erreur est propagée dans la hiérarchie de tâches, en remontant la relation **depends_on**. A chaque étape, des procédures spécifiques, appelées gestionnaires d'exceptions, sont appelés. Ces procédures doivent, si possible, réparer l'erreur. De plus, si au cours de la propagation de l'exception une tâche possède une tâche de planification, nous appelons les gestionnaires d'exceptions sur cette dernière, afin d'insérer les processus de planification dans le système de réparation de plan.

3.2.3 Réaction aux erreurs non gérées

Après cette phase de gestion d'erreur, le système régénère la liste de violations de contrainte restantes en rappelant les procédures d'analyse du plan. Les tâches qui sont marquées par les erreurs ainsi détectées sont tuées, ainsi que toutes les tâches qui en dépendent via n'importe lequel des graphes de relation.

En effet, ces erreurs soit sont des erreurs non récupérées par la phase de gestion d'erreur, soit des erreurs provenant des réparations elles-même. Dans les deux cas, il semble déraisonnable de rappeler une phase de réparation.

3.3 Garbage collection

Notre plan est construit autour de graphes de tâches. Parce que nous manipulons des graphes, il n'est pas pratique de demander que chaque tâche interrompe les tâches qui ne lui sont plus utiles : seule une analyse globale du plan peut déterminer quelles tâches doivent être interrompue car une tâche qui n'est plus utile à une activité donnée peut encore l'être pour une autre. Terminer ces tâches inutiles est le rôle de la troisième phase du cycle d'exécution : la phase de garbage collection.

3.3.1 Notion de tâche utile

Les tâches d'un plan sont séparées en trois parties :

– les tâches qui sont marquées comme utiles, comme par exemple les missions qui ont été données au système.

– les tâches qui permettent de réaliser les tâches de la première catégorie.

– le reste des tâches, qui ne sont pas ou plus utiles au système.

Une tâche est marquée comme utile si elle fait partie d'un de ces quatre ensembles :

– l'ensemble de *missions*, qui est l'ensemble des tâches que le robot tente de réaliser.

– l'ensemble des *tâches permanente*, qui est l'ensemble de tâches qui ne doivent pas être interrompues automatiquement.

– l'ensemble des tâches utilisées dans les plans en cours de construction.

– l'ensemble des tâches utilisées dans une interaction. Cette catégorie sera décrite plus en détail dans la prochaine section.

L'ensemble des tâches pour lesquelles une erreur non gérée a été détectée est toujours inclus dans l'ensemble des tâches à interrompre.

3.3.2 Interruption automatique de tâches

Notre système interrompt tâches inutiles incluses dans le plan en commençant par le haut du plan : le système ne doit pas terminer une tâche si d'autre tâches en dépendent encore. Les tâches qui n'ont de parent dans aucune relations sont donc interrompues en appelant leur évènement e_{stop} si il est contrôlable. Lorsque ces tâches sont terminées – donc potentiellement dans des cycles d'exécution suivants – le système tue les tâches qui n'ont plus de parents et ainsi de suite.

Une tâche dont l'évènement e_{stop} est non contrôlable est de toute évidence pas considérée par ce mécanisme d'interruption. Toutefois, si cette tâche est l'origine d'une erreur, ou si la commande de l'évènement e_{stop} échoue, la tâche est mise en quarantaine : elle n'est plus sujet à garbage-collection, mais elle ne peut plus non plus être utilisée par le reste du plan.

3.4 Exécution distribuée

3.4.1 Communication avec d'autres gestionnaires de plans

Quand ils sont en interaction, les gestionnaires de plans maintiennent deux états :

– ils sont en communication si un lien de communication existe entre les deux composants.

– ils sont *connectés* si ils interagissent. Deux gestionnaires de plans peuvent être connectés et n'avoir pas de lien de communication. Toutes parties d'un plan qui dépend d'un gestionnaire de plan distant est bien évidemment dépendante de cette connexion.

Quand un gestionnaire local est connecté à d'autres gestionnaires, il notifie ces gestionnaires distants des modifications apparues dans son plan (structure et exécution). Les messages ne sont envoyés que pour les parties du plan pour lesquelles le gestionnaire distant est souscrit.

Afin de représenter la dépendance du plan joint aux connexions, une tâche spécifique est insérée dans le plan pour tout gestionnaire de plan distant, et toutes les tâches qui appartiennent à ce gestionnaire sont **executed_by** cette tâche. Ainsi, si la connexion est perdue – par exemple parce que le gestionnaire local a pu déterminer que le gestionnaire distant n'est plus capable de remplir ses engagements – l'évènement e_{failed} de cette tâche est émis et les tâches correspondantes sont terminées.

3.4.2 Gestion d'évènements joints

La gestion d'un évènement joint est basé sur les règles suivantes :

1. la commande d'un évènement joint doit être appelée sur tous les gestionnaires de plan à qui appartiennent cet évènement.
2. l'évènement n'est émis que lorsque tous les gestionnaires de plans à qui il appartient ont annoncé qu'ils pouvaient l'émettre.

Ces deux règles permettent de représenter les mécanismes de la théorie des intentions jointes de Cohen et Levesque via le système de tâche/évènement. Ainsi, une tâche jointe peut être vue comme un *but joint persistant* :

– tous les robots pensent que la tâche n'est pas encore achevée

– ils ont tous acceptés d'accomplir la tâche jointe.

– ils vont tous tenter de l'accomplir tant qu'ils ne sauront pas qu'elle est accomplie ou qu'elle ne peut pas être accomplie.

3.4.3 Différences à l'exécution entre plans mono et multi-robots

Lorsque le système manipule des plans joints, quelques différences de comportement existent dûs à la perte du caractère synchrone de mécanismes de l'exécution : propagation des évènements, propagation des exceptions.

De plus, le système de garbage collection doit être capable de gérer les tâches qui ne sont plus directement utiles pour le gestionnaire local, mais le sont pour les gestionnaires distants.

3.5 Résumé

Ce chapitre a présenté les mécanismes qui rentrent en jeu au cours de l'exécution de du plan. Le cycle d'exécution est basé sur trois phases :

1. la phase de propagation d'évènements, où un algorithme de propagation globale permet de répondre aux évènements extérieurs.
2. la phase de gestion d'erreur, où les erreurs détectées pendant la phase de propagation, et les violations de contraintes présentes dans le plan, sont traitées.
3. la phase de garbage collection, ou un algorithme d'analyse globale du plan termine les tâches qui ne sont plus utiles pour la réalisation des buts du système et les tâches pour lesquelles des erreurs ont été détectées.

4

Gestion de plans

Par “gestion”, nous entendons la capacité de modifier le plan en cours d'exécution. Ce chapitre présente d'une part les *transactions*, qui sont un mécanisme central pour la modification des plans dans notre composant. D'autre part, il présente deux opérateurs de modification du plan qui prennent en compte l'état courant d'exécution du système, démontrant le développement d'opérateurs de modification de plans plus complexes sur la base de notre modèle de plan et de notre système d'exécution.

4.1 Exécution et modification simultanée des plans

4.1.1 Motivation

Notre modèle de plan ne permet pas, à l'exécution, de vérifier si chaque tâche prise à part peut ou non être exécutée dans la situation courante. En effet, notre gestionnaire de plan s'appuie sur les relations entre tâches et entre évènements pour déterminer cela. Il est donc critique que le plan vu par l'exécutif soit toujours complet : il ne doit pas y manquer d'objets ou de relations.

Cette propriété doit être maintenue alors que le plan est modifié. À cette fin, nous avons développé un mécanisme central dans notre système de gestion de plan : la transaction.

4.1.2 Représenter les modifications du plan

Une transaction est une représentation d'un ensemble de modifications qui permettront de transformer le plan courant en un nouveau plan, lui aussi complet. Cette idée est empruntée au monde des bases de données, adaptée à notre problématique de gestion de plan. Dans notre système, les transactions contiennent l'ensemble des modifications nécessaires pour transformer le plan courant en un nouveau plan. Ces modifications peuvent être appliquées au plan courant

quand la transaction est complète via une opération de *commit*. Si la transaction n'est plus valide ou plus nécessaire, elle eut être abandonnée (opération de *discard*).

4.1.3 Gestion de conflits entre exécution et modification du plan

La construction d'un nouveau plan est une opération potentiellement longue. Si nous voulons que notre système puisse évoluer pendant que de nouveaux plans sont construits, il est nécessaire de gérer les conflits pouvant apparaître entre les modifications du plan dues à l'exécution est les modifications représentées dans les transactions.

Nous définissons un ensemble de conflits pouvant ainsi apparaître, et nous définissons un cycle d'interaction entre le producteur de plan qui construit la transaction, l'exécutif et un composant à part, appelé le contrôle de décision.

4.1.4 Transactions comme outils distribués de modification de plan

Le mécanisme des transactions est très bien adapté à la modification de plan distribuée : chaque robot peut dans une transaction librement modifier son propre plan et les plans de ses pairs. De plus, une transaction distribuée ne peut être appliquée au plan que lorsque tous les gestionnaires de plans qui la gèrent l'acceptent. Si un consensus ne peut être atteint, la transaction est abandonnée. Les transactions sont donc un outil central en multi-robot pour négocier en se basant sur les plans.

4.2 Modifier le plan

4.2.1 Notion d'appartenance et modification directe du plan en cours d'exécution

Lorsqu'une relation est ajoutée entre deux tâche, ou lorsqu'un signal est ajouté entre deux évènements, une contrainte est ajoutée sur le gestionnaire de plan à qui appartient les tâches ou les évènements. Dans notre système, nous avons voulu que de tels ajouts de contraintes ne puissent être réalisés que via une phase de négociation. Le gestionnaire de plan interdit donc ce type de modifications.

À l'inverse, tout gestionnaire de plan peut enlever une relation si celle-ci s'applique sur au moins un objet qui lui appartient. En effet, un gestionnaire de plan doit garder autorité sur le système : il doit pouvoir, si nécessaire, se retirer de toute interactions qui le contraint.

4.2.2 Échange de sous-plans

Comme nous l'avons décrit en section 2.1.3, notre modèle de plan permet de représenter le fait qu'une tâche peut en remplacer une autre. Nous décrivons ici cette opération, ainsi que ses possibles interactions avec l'exécution.

4.2.2.1 Échanger deux sous-plans

Deux opérateurs sont définis :

– l'opérateur `replace_task` remplace simplement une tâche par une autre dans toutes les relations dont faisait partie la tâche d'origine. Cela permet, par exemple, de relancer une tâche qui a échoué, le reste du plan restant tel quel.

– l'opérateur `replace_plan` remplace une tâche ainsi que tous le sous-plan de cette tâche. Cet opérateur permet de remplacer, par exemple, une modalité de déplacement par une autre.

De plus, l'état d'exécution de la tâche de remplacement doit être comparé à celui de la tâche remplacée. En particulier, une tâche en cours d'exécution doit être remplacée par une tâche en cours d'exécution. Nous utilisons à cette fin une routine spécifique à chaque modèle de tâche, chargée de rendre les deux états d'exécution compatibles.

4.2.2.2 Gestion d'échanges non instantanés : transitions

Dans les cas simples, il est possible d'amener la nouvelle tâche à l'état d'exécution désiré, puis de réaliser le remplacement. Toutefois, dans certains cas cela est impossible car les deux tâches sont en conflits et ne peuvent s'exécuter en même temps. Le mécanisme de *transition* autorise de telles modifications en prenant en compte l'opération de remplacement lorsque la tâche d'origine est stoppée.

4.2.3 Interrompre et reprendre des activités

L'opérateur `split` "découpe" un plan en deux parties : une partie où un certain nombre d'activités sont interrompues, et une partie où elles sont relancées. Cet opérateur prend en compte l'état courant des tâches en question, et peut également prendre en compte des routines spécifiques à certains modèles de tâche.

4.3 Résumé

Ce chapitre a présenté l'outil central définit par notre système pour la modifications des plans : la transaction. Cette outil permet de modifier le plan du système pendant qu'il est exécuté, dans des contextes mono et multi robot.

Nous avons également présenté deux exemples d'opérateurs de modification du plan. Ces opérateurs démontrent la faisabilité de construire des mécanismes plus complexes sur la base de notre modèle de plan et de nos mécanismes d'exécution.

5

Implémentation et résultats

5.1 Implémentation : développement d'un contrôleur Roby

L'implémentation actuelle de notre système a été réalisée dans le langage de programmation Ruby. Notre choix s'est porté vers ce language car c'est un langage orienté-objet qui permet de manipuler les classes comme des objects, et cela a eu un impact intéressant pour la partie de notre système qui doit manipuler les modèles de tâches.

5.1.1 Définition de tâches et d'évènements

Les relations entre modèles de tâches se marient bien avec les notions propres au paradigme orienté objet : les modèles sont des classes et l'héritage permet de représenter la hiérarchie de modèles que nous avons présenté. Dans notre implémentation, nous avons pu utiliser les capacités d'introspection et les capacités de metaprogramming pour ne pas avoir à définir un langage de définition à part : les modèles, procédure et les bibliothèques de gestion de plan sont écrits directement dans le même langage. Cela nous a donné une grande flexibilité vis-a-vis de l'extension de notre système.

5.1.2 Contrôler GenoM depuis Roby

La couche fonctionnelle du rover Dala est fait d'un ensemble de modules fonctionnels GenoM. Afin de contrôler cette couche fonctionnelle depuis notre gestionnaire de plan, nous avons développé une couche de compatibilité entre Roby et Genom, permettant de représenter les activités des modules GenoM par des tâches dans le plan. Le développement de cette couche a permis par ailleurs de démontrer la simplicité d'intégration d'un outil comme GenoM dans notre système.

5.1.3 Test d'applications Roby

Le test est une composante essentielle du développement d'une application robotique. Afin de faciliter cette étape, nous avons intégré dans notre système un cadre de tests unitaires, dans lequel nous avons été capable de mettre en place des tests à plusieurs niveaux de détail :

- test unitaire d'un unique service de la couche fonctionnelle ;
- test d'intégration de plusieurs services ;
- test de la génération de plan ;
- simulation intégrée ;
- tests sur le terrain ;

Le maître mot est ici l'intégration : afin de permettre l'utilisation d'ensembles de tests, il est nécessaire de pouvoir lancer et quitter de manière automatique une application complète. Cela nécessite un grand niveau d'intégration des différents outils composant le système, niveau offert par notre applicatif.

5.1.4 Performance

Sur notre rover, le temps moyen d'un cycle d'exécution est d'environ 10ms pour une moyenne de 55 tâches. Des problèmes de latence liés au garbage collector de l'interpréteur Ruby lui-même nous a toutefois poussé à fixer ce cycle à 50ms, ce qui est largement suffisant dans le cas de notre application : les réactions qui doivent être faites en temps borné doivent être implémentés dans la couche fonctionnelle GenoM.

5.2 Expérimentation

5.2.1 Supervision du rover Dala

La supervision de notre rover a permis d'utiliser notre système extensivement, et en particulier d'utiliser les mécanismes de reprise d'erreur

Cette section présente deux des points principaux liés à la supervision de Dala :

– le système doit prendre en charge l'activation des cycles de mise à jour de la carte de terrain et de la carte de traversabilité. En pratique, cela est réalisé en utilisant deux outils

- des évènements lié à l'état du robot : changement de position supérieur à une certaine valeur, timeout, ... Ces évènements peuvent être composés par des opérateurs *et* et *ou*.
- un gestionnaire de boucle, une tâche de planification qui permet de développer dynamiquement des séquences d'une même action.

– le système de navigation nécessite plusieurs gestionnaires d'erreur, utilisant à la fois les exceptions et la relation **error_handling**.

5.2.2 Résultats

Le scénario complet a été implémenté et testé en simulation. Par ailleurs, supervision du rover Dala a également été testée sur le terrain. Notre système de simulation permettant d'utiliser

inchangés la plupart des modules du robot réel, le passage à l'expérimental n'a posé aucun problème majeur du point de vue du système de supervision.

La simulation est basée sur un terrain de 400x500m. Deux cartes de traversabilités sont générées : une pour simuler la perception à haute altitude et une pour simuler la perception à basse altitude.

5.3 Résumé

Notre implémentation et sa validation expérimentale ont été d'une grande importance au cours du développement des concepts présentés dans ce manuscrit. Ainsi, la gestion des deux robots ainsi que la gestion de leur interaction a mis en exergue plusieurs caractéristiques de notre approche :

Transactions Toute génération de plan dans Dala est faite de manière asynchrone, en utilisant des transactions. Cette caractéristique s'est montrée très utile par exemple lorsque le plan échoue au cours de son adaptation. De plus, l'utilisation des notifications et des transactions distribuées pour la gestion de l'interaction entre les deux robots s'est montrée être une approche élégante.

Gestion centralisée de toutes les activités du robot Cette caractéristique s'est également montrée très importante lors du développement de nos deux robots : elle a permis une robustesse et une flexibilité certaines lorsque des problèmes sont apparus dans le logiciel en cours de développement.

Exensible La mise en place de ce scénario a nécessité le développement d'outils de plus haut niveau, directement basés sur notre modèle de plan et sur notre mécanisme d'exécution. Ces outils ont montré que notre système pouvait supporter le développement de mécanismes plus complexes.

6 Conclusion

6.1 Résumé

Le composant de gestion de plan présenté dans cette thèse a été conçu pour le contrôle basé sur des plans d'équipes de robots. Ce composant répond aux besoins suivants :

– un modèle de plan permettant la traduction de plans provenant d'autres planificateurs, et assez riche pour permettre l'intégration de mécanismes de supervision et d'adaptation complexes.

– un mécanisme d'exécution pour ce modèle.

– des outils génériques pour l'adaptation de plan. Principalement, le mécanisme de *transaction* permet d'adapter et d'exécuter simultanément le plan en prenant en compte les conflits entre l'exécution et les plans en cours de construction.

Cet outil logiciel a été testé en simulation pour la partie multi-robot et sur le terrain pour la gestion du robot seul.

6.2 Perspectives

Notre système profiterait grandement d'une représentation explicite du temps : d'une part de la prédiction de l'instant d'émission des évènements, et d'autre part de l'intégration de réseaux de contraintes temporels. La prédiction d'état est également une extension assez évidente à un système comme le nôtre.

En se basant sur ces deux extensions, l'intégration d'un mécanisme d'ordonnancement permettrait de profiter pleinement du fait que notre système représente toutes les activités du système, y compris les tâches de planification.

Enfin, un mécanisme de plan merging permettrait une intégration plus directe de plusieurs plans dans le plan commun.

De manière générale, l'intégration de plusieurs systèmes de décision dans un même robot est à la fois une nécessité et un problème difficile. Alors que l'intégration des couches fonctionnelles est un sujet abondamment traité par la littérature, l'intégration d'outils décisionnels l'est peu. L'intégration de tels outils, mais également d'outils de diagnostic et de reprise de fautes, dans un système centralisé de gestion de plan comme le nôtre serait à la fois une avancée significative pour la robotique, mais permettrait également – à plus court terme – une plus grande réutilisation des différents outils logiciels.

Cette intégration serait par ailleurs certainement facilitée par une formalisation de notre modèle de plan et des mécanismes que nous avons construit autour de lui.

Author: Sylvain Joyeux

Title: A Software Framework for Plan Management and Execution in Robotics: Application to Multi-Robot Systems

Abstract: During the 90s, the integration of the many functionalities needed to make robot autonomous has given birth to robotic architectures, which allow cooperation between perception, decision and action in robotic systems. Experience with these architectures has shown that they suffer from limitations. More recently, new paradigms have appeared to tackle these limitations, based mainly on the idea that plan representation should be unified. This thesis contribution is a plan model which allows the integration of the result of different decision formalisms, to execute them and to adapt them online. Moreover, this model and the execution and adaptation component built around it have been designed with multi-robot in mind: it allows to build, execute and adapt joint plans, in which more than one robot are involved. The software component written during this thesis has been tested experimentally, in an aero-terrestrial cooperation scenario.

Keywords: architecture, planning, multi-robot

Auteur : Sylvain Joyeux

Titre : Un composant logiciel pour la gestion et l'exécution de plan en robotique : Application aux systèmes multi-robots

Directeurs de thèse : Simon Lacroixet Rachid Alami

Thèse soutenue le 6 Décembre 2007au LAAS/CNRS, Toulouse, France

Thèse préparée au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS/CNRS)
7 av du Colonel Roche
31077 Toulouse Cedex 4

Discipline : Systèmes informatiques critiques

Résumé : Dans les années 90, le problème de l'intégration des nombreuses fonctionalités nécessaires à l'autonomie de robots a donné naissance aux architectures robotiques, qui permettent aux différentes fonctions nécessaires aux robots autonomes de bien s'articuler entre elles. la perception, la décision et l'action. L'expérience dans ce domaine a montré les limites des différentes approches alors proposées. Récemment, de nouvelles architectures ont tenté de dépasser ces limites, principalement en unifiant la représentation du plan. Cette thèse propose à la fois un modèle de plan permettant de représenter les résultats de différents formalismes de décision, d'exécuter le plan qui en résulte, et de l'adapter en ligne. Ce modèle et le composant d'exécution et d'adaptation construit autour de lui ont été pensé dès l'origine pour le multi-robot : il s'agit de permettre l'exécution et l'adaptation de plans joints, c'est à dire de plans dans lesquels plusieurs robots coopèrent. Le composant logiciel construit durant cette thèse a de plus donné lieu à une validation expérimentale pour une coopération aéro-terrestre

Mots-clés : architecture, planification, multi-robots