A plan manager for multi-robot systems

Sylvain Joyeux, Rachid Alami, Simon Lacroix, Roland Philippsen

A plan manager for multi-robot systems

Sylvain Joyeux[1,2], Rachid Alami[1] and Simon Lacroix[1] and Roland Philippsen[3]

[1]LAAS-CNRS, Université de Toulouse
sylvain.joyeux@m4x.org, simon@laas.fr, rachid@laas.fr

[2]Currently affiliated with DFKI Robotik Lab, Bremen

[3]Artificial Intelligence Laboratory, Stanford University
roland.philippsen@gmx.net

Abstract

This paper presents a software component, the plan manager, which provides the services needed to build and execute plans in a multi-robot context. This plan manager handles fully dynamic plans (insertion and removal of tasks), provides tools for safe concurrent execution and modification of plans, and handles distributed plan supervision without permanent robot-to-robot communication. The proposed concept is illustrated by a scenario which involves the navigation of a rover and an UAV in an initially unmapped environment.

1 Introduction

The experience gained in the last ten years of robotic architectures can lead to the definition of a generic architecture where the place for integration is a single *plan management component*: the integration of decision-making is done through the plan object, which has to be revised continuously during execution. This section analyzes the main existing approaches for single robot and multi-robot planning architectures, introduces the notion of plan management through previous approaches that use the same central idea, and outlines the rest of the paper.

1.1 Hybrid architectures for mono and multi-robot systems

Hybrid architectures, in which a reactive layer is coupled with a decision layer, have shown to yield efficient intelligent behaviors. Within such architectures, there is a need to integrate different kinds of decision making tools centered on anticipation for the following three reasons:

- there is no universal planning model and planning algorithm. One should therefore select the right model and tool for a specific application and/or situation. Planning is therefore a decentralized operation within a robot – and even more in a multi-robot system. A successful all-purpose robotic architecture should therefore allow for a dynamic selection and configuration of the decision-making tools.

- the planning problem is hard to compute. Generating the plan for a whole robot mission – and even more for a robot team – is hardly tractable. Thus architectures must allow to split the problem of building the whole system plan into smaller problems, which can be managed by plan generation tools, and then to execute the set of sub-solutions in a coherent way.
- humans are to be integrated in the decision-making process: the robot highest-level goals are defined by humans, and humans may have wider knowledge than the robots. Integrating humans in the decision-making *and* in the plan execution loops yields to more efficient robot systems.

A strong trend in the robotic architecture community has been to make the representation of the plan follow the separation of planning tools: pairings of planning tool/execution engine are defined, each of them only knowing its own plan and a small subset of the other's. Examples of this kind of approach is the LAAS architecture [Alami et al., 1998a] where planning/supervision pairs are stacked, or the IDEA architecture [Muscettola et al., 2002] in which planners/execution engine pairs are integrated in a network.

Both approaches share a common problem: they do not exhibit any representation of the whole system plan. This is because these architectures are centered on *planning*: since planners cannot generate the whole system plan, there is no point in representing it. Moreover, the various planners involved may not even employ the same model, so it is not possible to merge their result in a global plan. Splitting the global plan in various models forbids the integration of *generic* plan analysis tools, *e.g.* for diagnostics or state projection.

For multi-robot systems, the first trend has been to adapt existing mono-robot architectures to the distributed nature of the problem. Such an adaptation has been made along two main schemes:

- On the one hand, some approaches rely on a team-management layer which sends orders to a mono-robot supervision system (for instance Teamwork [Tambe, 1996] and TraderBots [Dias, 2004]). Such a scheme cannot pro-actively anticipate interactions within the plan structure, because interactions between robots are handled reactively. Also, the team-management layer is limited to managing high-level tasks, which makes it difficult to handle tighter interaction modalities, like opportunistic cooperation, which take place at a lower abstraction level.
- On the other hand, some approaches rely on the coupling of a team-management layer on top of a modified mono-robot supervision systems that can send orders transparently to other robots: FIRE [Simmons et al., 2002] is for instance a modified TDL system coupled with a Contract-Net Protocol [Smith, 1980] task allocation scheme. The resulting system has two flaws: first, the modified TDL system, in which a robot can change the state of a remote robot, does not act pro-actively – it does not act on plans. It therefore assumes that communication is available at all times since one robot cannot act on the basis of the predicted behaviour of another robot. The second flaw is the same as for the layered mono-robot architectures: the two layers do not share all the information they have, and so the interaction managed by the upper layer can only loosely take into account the interaction in the lowest one. A common place in which all the interactions in place are described is missing.

Of course, other architectures have also been defined from the ground up with multi-robot aspects in mind. One can distinguish four important plan-based approaches in the

literature: the M+ architecture [Botelho and Alami, 1999], GPGP and its associated task representation TAEMS [Lesser et al., 2004], the COMETS architecture [Gancet et al., 2005] and Machinetta [Schurr et al., 2005].

The M+ architecture is based on mapping a planning/executive pair on one robot onto a corresponding pair on another, and therefore limits the interaction possibilities to the subplans each pair currently know. In COMETS, a plan which can contain joint tasks[1] is produced by a Shop2 planner. The joint tasks are then negotiated among the team by an "Interaction Manager". The resulting plan (developed joint tasks and mono-robot tasks) is then sent as a unique plan to an execution component. In GPGP, the joint plan is modified through a whiteboard mechanism by all involved robots. When they all agree on the result, each robot sends it to its own scheduler which starts its execution, the execution mechanism being able to take the specificities of robot-to-robot interactions into account. Machinetta is based on teamwork mechanisms, in which proxy agents represent each member of the team. These proxy agents manipulate a team plan, but with only a partial knowledge of the plan of the member they represent. The main contribution of Machinetta is that, based on this team plan, each proxy agent is able to act upon the predicted behaviour of the other members of the team *without explicit communication*. The main focus of COMETS and Machinetta is to define a generic team management system and the focus of GPGP is multi-robot planning and scheduling.

All these approaches are tied to their particular interaction scheme: none has been designed to be flexible enough to integrate different plan production mechanisms and different team management schemes.

1.2 A plan management approach

We advocate that the integration of the various tools centered on plans – planners, plan analysis and interaction mechanisms – should be done through the construction and adaptation of a representation of the whole system plan. We therefore introduce a *plan management component*, whose role is to allow the construction, execution and continuous adaptation of such a plan, either in mono-robot or multi-robot contexts. This component is not centered on planning *per se*. Instead, it allows integrating the aforementioned tools, by providing a plan model, an execution scheme, and related basic services.

The plan management component is the place where various interaction protocols can be integrated: it relies on the definition of *(i)* a plan model in which the various approaches of multi-robot plan representation are integrated, *(ii)* an execution scheme for this plan model, and *(iii)* a plan modification tool which can be used to integrate various negotiation protocols based on plans.

This component is also a mean to integrate humans in the loop. Humans may want to change plans locally, but without being able to grasp the consequences of those small changes on the whole system plan. This is one of the focus of mixed-initiative planning ([Bresina et al., 2005, Myers, 1996, Myers et al., 2003]): allowing a user to change an automatically generated plan, while being able to represent the consequences of these changes on the whole plan, and – if needed – the incompatibilities between the user's decision and the plan itself. In our approach, this issue can be tackled at the planner level if the planner allows user integration, but especially at the plan level: since our system represents the whole system plan, it is able

[1] A joint task is a task which is executed by more than one robot at a time

to represent the consequences of the user's change on this plan. From the plan manager point of view, the user is only an additional decision-making agent. As for human intervention on the executed plan, it becomes a problem of *authority management*: in case of conflicts, the system must be able to decide whether the user change is of higher priority than the previously taken decisions.

1.3 Related work

This kind of approach, centered on the plan object, is something which has already been advocated in the Claraty architecture [Volpe et al., 2001], for the development of their CLEaR component, which was originally envisioned as the place where a Casper [Chien et al., 2004] planner and a TDL [Simmons and Apfelbaum, 1998] executive could interact via the representation of the whole system plan. However, it seems that this approach has been abandoned since. Another plan-centric approach is the Concurrent Reactive Plans of Beetz et al. [Beetz, 2000, Beetz et al., 2001]. From Beetz's point of view, such a component is a way to make the use of plan libraries robust in the real world: he developed the idea of *transformation planning* as a way to adapt plan libraries to the execution context, and integrated generic plan-based methods to detect plan flaws through state projection and forestall them through proactive plan adaptation. From our point of view, this body of work shows that it is possible to build a system in which, *given a well-formed plan*, there are tools to adapt this plan continuously, regardless of the tool which generated them. Thus, it is possible to integrate rich schemes in a generic plan management component.

These few approaches, while working directly on the idea of a plan, are not centered on plan management. As the authors of [Pollack and Horty, 1999] describe it, plan management is more than having a plan itself. According to them, the *raison d'être* of a plan management system is the following functionalities:

- plan generation: being able to produce plans
- commitment management: being able to decide what to keep and what to drop, in light of new information and new goals.
- environment monitoring: on one hand, check that the current state of the world is compatible with what is needed for your own plan. On the other hand, also monitor the conditions that led you to *reject* other opportunities.
- alternative assessment: the ability to take decisions between exclusive alternatives (alternatives where you have to chose one over the others).
- plan elaboration: view the plan production process as a dynamic process itself, allowing executing unfinished plans, while constraining the planning process to be sure that plans are available "at the right time".
- coordination with the other agents: being able continuously build and execute distributed plans, in an environment where communication cannot be taken for granted, and where agents can take individual decisions that go against the current state of the multi-robot plan.

The authors then presents the Plan Management Architecture (PMA), as an example of an architecture partially answering these points. This architecture is designed to be a plan-management assistant for human operators (smartly managing the schedules of the humans). In our opinion, this architecture is not suitable as-is for robotic purposes, particularly for

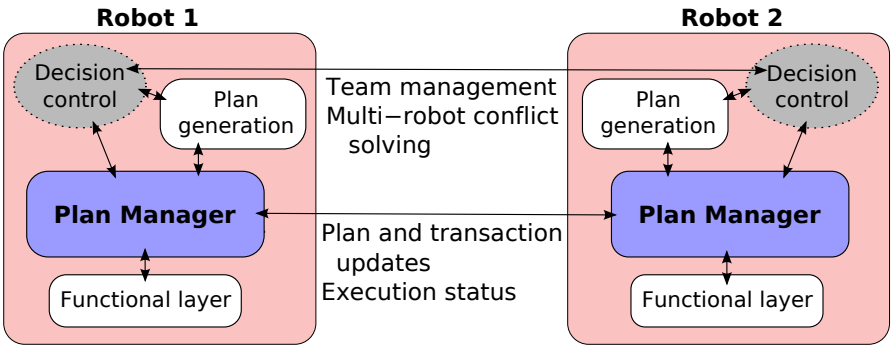


Figure 1: In this architecture, each robot plan is managed by its own plan manager. Parts of these plans are shared among plan managers. Decision control manages plan generation, team management and solves conflicts between execution and planning

multi-robot systems, as it does not take into account fundamental issues such as continuous planning and execution, failure representation and monitoring, and possible communication problems. A follow-up work of PMA, the Continuous Planning and Execution Framework (CPEF, [Myers, 1998]) has improved on these points in the framework of HTN plans. Because of the choice of this model, this work lacks the ability to represent in its plans the relations between parallel activities as GPGP/TAEMS (HTNs represent only refinement relations, where child actions are less abstract representations of their parents). This ability is, in our opinion, fundamental in multi-robot systems.

1.4 Outline

The next section gives an overview of the plan management component. It defines its structure and integration within an hybrid architecture, and presents the basics of the plan representation and of the mechanisms that manipulate plans to ensure their consistent execution and modification. The section then introduces a supporting scenario that involves the cooperation of a rover and a UAV, which is used as an illustration throughout the paper, and has been implemented both in simulation and on real robots using our software component. Section 3 depicts the plan representation, and sections 4 and 5 respectively depicts the plan execution and adaptation mechanisms. Section 6 finally illustrates the achievement of the rover/UAV scenario.

2 Approach overview: an architecture with plan management at its core

2.1 Role of the Plan Manager

The role of our plan manager component is to provide the services required to represent, build, execute and adapt multi-robot plans. The component is generically designed: it is not associated to specific plan generation systems or specific team management schemes. This plan manager acts as a broker between the functional layer and plan generation components (Fig. 1). In this architecture, the planners are responsible for producing coherent plans

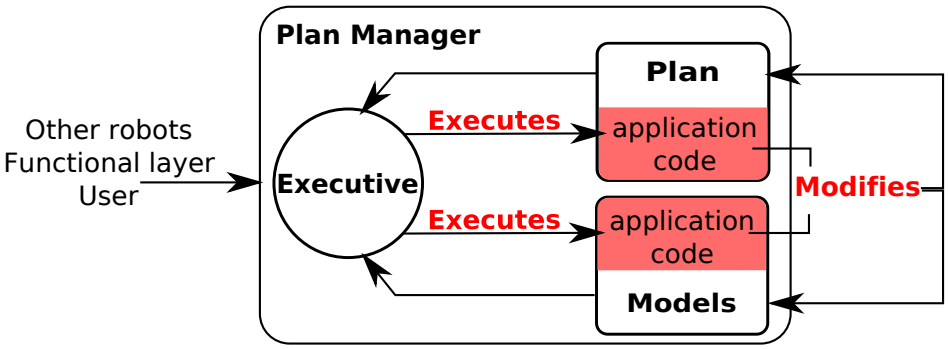


Figure 2: Composition of the plan management component: the executive calls application code based on the plan and a set of external events. This application code can then *adapt* the plan and the models.

either for the robot alone or for the team, and the functional layer provides services such as perception and action algorithms that interface the robot with the real world. Between these two, we introduce two components: the *plan manager* maintains a plan, which is a graph of tasks and events defining what the robot may do in the future and how it will do it. The plan manager is also responsible to execute this plan, which means that it sends commands to the functional layer and interprets its feedback information according to what is expected in the plan. Then, the *decision control* component has two roles: first, it may call the planners for instance to adapt the plan for new missions, for contingency planning and because of cooperation opportunities with other robots. Second, it handles the choices that have to be made during execution: since our system allows simultaneous planning and execution, conflicts will arise between the executed plan and the partial plans being built. In a multi-robot context, it also solves conflicts between the needs of the robot and the needs of the team.

2.2 Structure of the Plan Manager

The plan manager itself is based on the definition of various objects (Fig. 2):

- a set of *models*, based on a generic plan model designed for the representation of multi-robot execution situations.
- an *executive* that ensures the proper execution of plans – and in particular of joint plans.
- a way to safely *modify* plans as they are executed.

Plan model. The plan model is designed with *supervision* in mind: its goal is not to offer a planning model, but a model rich enough to represent the interactions between systems, for execution and situation assessment (in particular error representation and recovery). Unlike other plan-based systems, our model is neither solely based on a combination of tasks (like what can be found on HTN-like or procedural models like CRPs [Beetz et al., 2001]), nor based on scheduling constraints (temporal ordering of the beginning and end of tasks). Instead, our plan model separates the information into two dimensions:

- the activity information is represented by *tasks*, which are structured through a set of typed *relations*. There is no distinction, like in HTNs, between an abstract level and an

atomic level. Instead, all tasks are representation of activities running in parallel. The task structure is then – alike to the GPGP/TAEMS model – what defines the various interactions between tasks, as for instance dependency, influence, ...

- the temporal information is represented by two objects. *Events* represent the particular situations that are useful for the plan execution. *Event graphs* then represent the desired reaction to these events.

One singular property of our model is that the task relations are directed acyclic graphs (DAGs) of tasks. It has to be compared to existing systems where tasks are maintained in trees. In multi-robot systems, this DAG structure makes the representation and execution of plan-merging results natural, allowing for the exploitation of opportunistic situations. For instance, in [Clement and Durfee, 1999b, Clement and Durfee, 1999a] HTN-like plans are synchronized to make multi-robot plans, but introduces for that explicit multi-robot primitives. Their work do not deal with the execution and error handling of the resulting plan, but it is clear that the same mechanisms that could be implemented in each separate plan would not be applicable in the merged plan: the structure of the merged plan is not simply a merge of both plan structures (the refinement relations coming from the HTN tree are replaced by cross-plan synchronization primitives).

Finally, a single plan manager can represent both its tasks and tasks that are executed/managed by other managers: it is possible to link events and tasks of two different plan managers by the same relations. In order to represent in the same plan tasks that are executed by the local plan manager, tasks that are executed by another plan managers and joint tasks, two means are available: the notion of role [Tambe, 1996], and the notion of ownership (who does what).

The executive. The role of the executive is to use the plan object to determine the desired system reaction to particular situations, and in particular to represent execution errors and react to them. For multi-robot systems. it must be able to handle the fact that robots cannot communicate at all times, and that lack of synchronization in the plan may lead to problems during plan execution. Execution is a cyclic process of three steps (Fig. 3).

Plan adaptation. Our plan manager defines *transactions*, a tool for robust plan adaptation. This tool provides the services required to build multi-robot plans, and to use plans as a basis for negotiation. In a plan manager, one can build plans that involve multiple robots, make these robots negotiate about the new plan (distributed plan modification) and, if they agree, commit them to the result.

2.3 Plan-based cooperation in a rover/UAV scenario

In order to illustrate the concepts associated with the plan manager (that are detailed in the next three sections), we outline here a scenario which has been realized in simulation, and partially tested on real robots.

In this scenario, the mission to achieve is a rover `MoveTo` task in an initially unknown terrain. For that purpose, the rover is endowed with algorithms which build a traversability map from perceived data (`Bitmap` module), and a now classical two-layered approach to generate motions: the `Nav` module plans a long range path in the traversability map [Gancet and Lacroix, 2003],

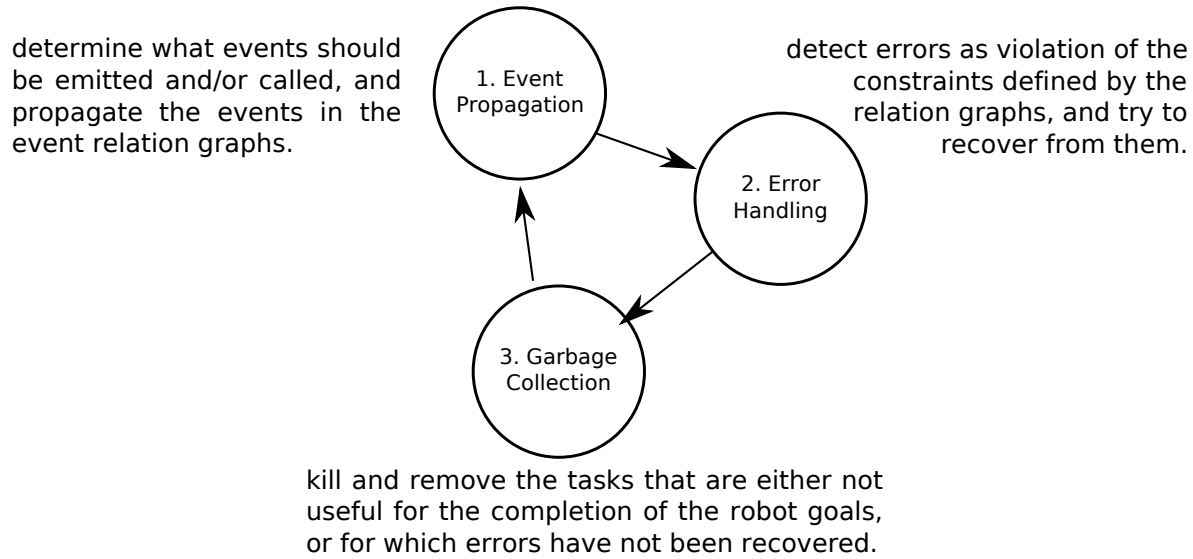


Figure 3: Overview of the execution cycle

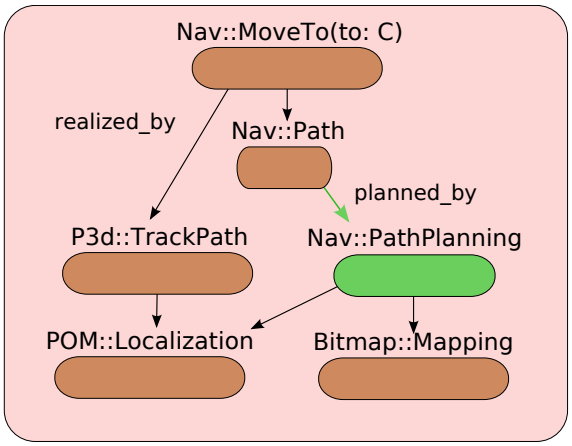
and the path is transmitted to the `P3D` module, a local planner that generates motion commands in an elevation map [Bonnafous et al., 2001]. This perception/decision/action loop is summarized in the plans by respectively the `Bitmap::Mapping`, `Nav::Path` and `P3d::TrackPath` tasks. The rover is assisted by an UAV, which builds traversability maps from vision data. We assume the UAV flies at an obstacle free elevation, so that its movement can be handled by a simple waypoint navigation scheme.

On the basis of this scenario, we express the rover-UAV cooperation as relations between individual robot plans, and then show how this joint plan is handled in our plan manager. Fig. 4 presents the initial rover plan, in which `P3d::TrackPath` follows `Nav::Path`, the waypoint list established by `Nav::PathPlanning` based on the results of `Bitmap::Mapping`[2]. The arrows between tasks express relationships which are described in section 3.1.

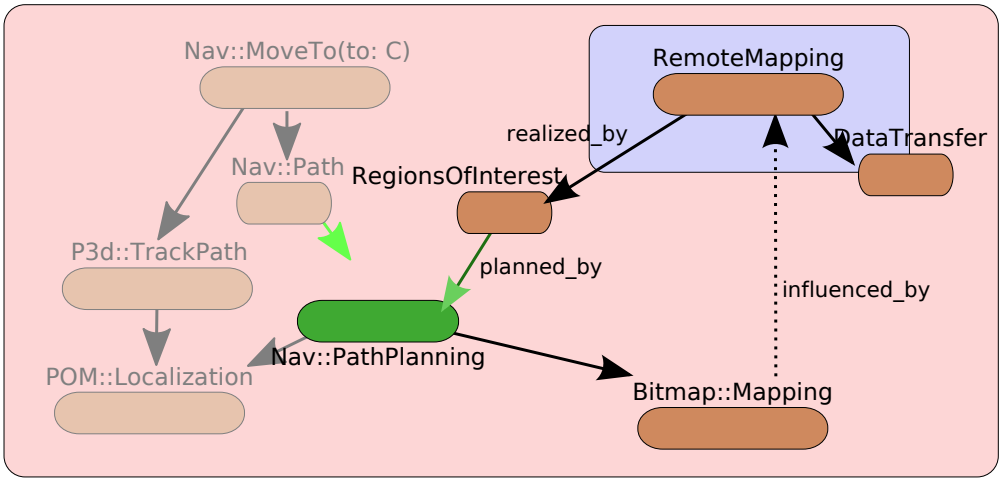
We implemented an opportunistic method to initialize the interaction for this scenario: the rover does not know beforehand that an UAV will help to realize its mission. When the UAV detects the rover by using an automatic network discovery mechanism, it adds *triggers* on the rover's plan. A trigger is a mechanism that allows a plan manager to be notified of changes to another robot's plan, making it possible for instance to initiate new interactions. We use a pattern matching approach that detects when certain tasks or relationships between tasks exist on the sending side of the trigger. When a trigger matches, the receiving plan manager is notified of this fact along with a description of the involved tasks and relations.

Fig. 5 shows an example with a trigger representing a situation upon which the UAV is able to act: as soon as the pattern matches, the rover sends the tasks to the UAV which integrates this information in its own plan. The UAV can then use a planner to adapt the plan for interaction, based on its partial knowledge of the rover's plan. This planner cannot directly change the current rover plan: first, partial plans are not sound (*i.e.* they miss for instance some activities or the representation of constraints), and second, if the UAV had such an ability, it would mean that the rover is not able to fully control its plan. We

[2] `POM::Localization` summarizes all the localization processes on-board the rover



Initial rover plan (partial view).



Final rover plan (partial view). The part with blue background is the partial view the rover has of the UAV plan. The `DataTransfer` task, which is in-between, is a joint task.

Figure 4: *(top)* partial view of the initial rover plan. *(bottom)* the cooperative plan built by negotiation. The interaction is based on rover-provided information on the regions which are of interest for its navigation. These zones are represented by the `RegionsOfInterest` task.

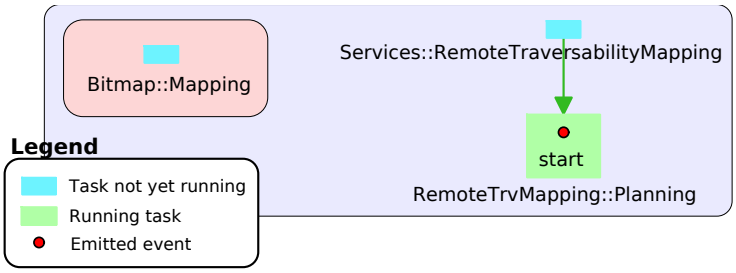


Figure 5: When the trigger matches, the UAV is notified of the corresponding tasks (`Bitmap::Mapping` task). Then, the UAV inserts two tasks: a `Services::RemoteTraversabilityMapping` task which is an abstract representation of its possible mapping activity and a `PlanningTask` which represents the negotiation activity required to build the joint plan. This negotiation activity is linked with the mapping activity through a specific relation which represents that `PlanningTask` generates the plan for the remote mapping.

therefore need a way to transform plans without directly changing the plan being executed, so that *(i)* the executive only sees complete plans and *(ii)* one robot can propose a plan modification to another. This is achieved by *transactions*, which represent a plan changeset, *i.e.* set of modifications required to get from the current executed plan to the desired final plan. Transactions can be shared and synchronized across plan managers.

So, while the UAV generates its proposal, the rover is not aware of the transaction. Once the transaction is complete from the UAV point of view, the transaction is sent to the rover. The rover can modify it, in which case the changes are sent back to the UAV: both robots use the transaction as a whiteboard to build their joint plan. Once both agree on the new plan, they change their executed plan accordingly at the same time, and can start its execution. If a joint plan cannot be found, the transaction is simply discarded.

3 Plan Model

Our plan manager has been designed with multi-robot systems in mind: in a multi-robot context, a single plan manager is able to express and manipulate plans where tasks executed by the local robot (local tasks) are interacting with tasks executed by other robots (remote tasks), or even the joint tasks, which imply more than a single robot. Note that in order to reduce plan complexity, there is no need for one robot to know *everything* about another's plan: a plan manager is informed only about the remote tasks it is interacting with (section 5). This allows to keep each plan at a tractable size regardless of the number of plan managers currently interacting.

This section first describes the plan manager model of tasks and events, and how their ¡¡¡¡¡¡¡ HEAD:main.tex structure forms a rich plan model. It then focusses on the multi-robot part of this ======= structure forms a rich plan model. It then focuses on the multi-robot part of this ¿¿¿¿¿¿¿ 86dd0dac22971a443f41129c4aa8099f33c64fc3:main.tex model: how the model represents which robot is responsible for doing what, and more particularly how team plans are represented.

3.1 Execution milestones: events

The basic element of execution is the *event*: the set of events in the system represent the observable situations during execution. To be represented in our system, the situations that are of concern for the robot supervision must have an associated event, or be built as a combination of other events (temporal combinators *and*, *or* and *until* are for instance built by using only the core event model). When the situation that is represented by a given event is reached, the event is *emitted*. Example of events are timer events (emitted periodically) or *task events*, which describe the progress of the task they are attached to – like the `stop` event of a task.

From a control point of view, events come into two categories:

- an event is *controllable* if the system can make sure it will be emitted. In this case, *event command* is the procedure able to make that happen.
- an event is *contingent* if the system has no direct control on its emission.

Moreover, specific procedures called *event handlers* can be attached to events. These procedures are called every time the event is emitted, allowing for reactions that are not encoded in the plan itself.

Finally, if an event cannot be emitted anymore (*i.e.* the underlying situation cannot be reached anymore), it is *unreachable*. This property is very useful for supervision as it allows to describe error conditions as the negation of a desired situation (*i.e.* an error occurs if a goal event cannot be reached).

3.2 Execution flow: event relations

With events, our plan model is able to represent the milestones of execution. The associated relation graphs then allow to represent the *execution flow*: how the system should react to a given situation.

Two relations are defined in our plan model:

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

From a semantic point of view, those two relations represent complementary things. The signal relation represents the system reaction: what actions the system should take when a given situation is reached. The forward relation represents event generalization: in $e_1 \xrightarrow{fwd} e_2$, e_2 is emitted in all situations where e_1 is – but can be emitted in situations where e_1 is not. The situations represented by e_2 are therefore a *superset* of the ones represented by e_1. This is used to classify situations, like the different fault modes of an activity which are forwarded to the generic `failed` event of the same activity, itself forwarded to `stop`. See Fig. 6 for a real-world example of the use of these event relations.

3.3 Activity representation: tasks

Similarly to TDL and other task-based systems, tasks describe the system activities. While events represent execution milestones, specific situations that are reached by the system, tasks represent the progressive activities that the system executes.

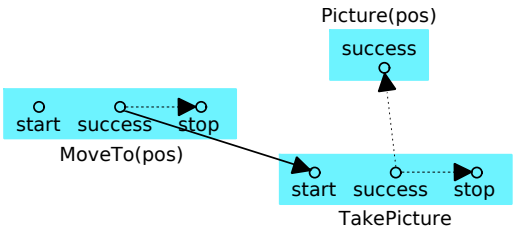


Figure 6: Example of event propagation. The `MoveTo` and `TakePicture` tasks form a sequence (`success` $\xrightarrow{sig}$ `start`). This sequence is the implementation of a higher-level task, `PictureAt`, which is why `success` is forwarded from `TakePicture` to `PictureAt`. This is a typical translation of a HTN-like hierarchy in our model.

In a very classic way, task models are defined by a *type* and a set of parameters needed to define the activity in a concrete way. For instance, `MoveTo(start, end)` or `Localization` are two task models. To be executable, a task model is instantiated by associating values to the parameters. The set of *running* tasks therefore represents parallel executions. These task models are put into a hierarchy (Fig. 7) in which the more abstract task models are refined in more concrete ones.

In order for the system to track each activity progression, each task model also defines a set of *events*, which are therefore milestones specific to the task execution. Example of such events are the standard `start` and `stop` events, but also model-specific ones like for instance a `blocked` event for a motion task.

In the plan, these models are used to define task *instances*, which are the *active* objects in the system: they determine when their events should be emitted, and they handle their event commands. These task instances can be *abstract* (*i.e.* non-executable): the initial rover plan is for instance a single, generic `MoveTo(x, y)` task, representing thus *what* the rover is planning to do but not yet *how*. For this activity to be executed, a specific `MoveTo` implementation must be chosen by the system. Task models are therefore forming hierarchies (Fig. 7) in which the more abstract task models are refined in more concrete ones.

Representing the relationships between models in this way has two advantages: it allows to determine that two activities are equivalent, based on their common ancestry. From a multi-robot point of view, it has the advantage that a robot does not necessarily have to know all the models known by all the plan managers. Instead, it can use an intermediate level of abstraction, without having any knowledge about another's specific task implementation. This has the advantage that the plan structure, as seen by the remote plan manager, is not changed. Only less information is represented as activities are represented in a more abstract way.

3.4 Activity structure: task relations

In a plan, tasks are structured in order to represent how one activity interacts with another. This has multiple roles:

- provide a semantic structure for the understanding of the plans, for instance to be interpreted by a human.

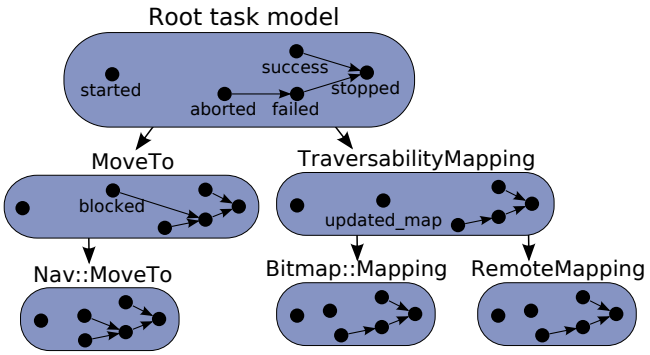


Figure 7: Task model hierarchy. Task models can inherit from other models, to express specialization of generic activities. On the right, we can see the abstract representation of a traversability mapping, which is implemented by the rover through the `Bitmap::Mapping` task and by the UAV through the `RemoteMapping` task.

- describe what effect one task has on another one, and what are the error conditions that are specific to each given relation. This information is more specific to the plan execution.

As we already stressed out, our activity structure is based on a set of directed acyclic graphs. This is a very important features which singles-out our system, since it allows to represent dependencies on common services like localization and locomotion. Moreover, in a multi-robot context, it allows to represent and manage ¡¡¡¡¡¡¡ HEAD:main.tex plans in which plan merging has been applied without representing local dependencies and cross-agents dependencies, contrary to prior work [Alami et al., 1998b, Yang, 1997, Clement and Durfee, 1999b, Clement and Durfee, 1999a]. ======= plans in which plan merging has been applied, representing local dependencies and cross-agents dependencies in the same way, unlike what exists in prior works [Alami et al., 1998b, Yang, 1997, Clement and Durfee, 1999b, Clement and Durfee, 1999a]. This has the noticeable advantage that the same supervision mechanisms (error detection and handling, plan adaptation) still apply to the multi-robot case. ¿¿¿¿¿¿¿ 86dd0dac22971a443f41129c4aa8099f33c

Therefore, a relation is defined by the following:

- a *parent* and *child* task
- a set of *requirements*, which are what the parent task expects from the child task in the context of this relation.
- a set of *error conditions*, which are what the parent considers undesirable situations in the context of the relation. This always includes the failure to meet the requirements.

In its current iteration, our system defines three relations:

- the `realized_by` relation is used to express hard dependency. In this relation, the parent task requires that one event in a given set $E_{success}$ of success events is emitted. A set of $E_{failure}$ events are the set of events that are undesirable. A $t_a \rightarrow t_b$ `realized_by` relation is therefore defined by

$$\text{realized\_by}(t_a, t_b, E_{success}, E_{failure})$$

For instance, on Fig. 4, `Nav::MoveTo` requires that its two child tasks are executed continuously. Therefore $E_{success} = \emptyset$ and $E_{failure} = stop$. The refinement relation used by HTN-based systems is a `realized_by` relation with $E_{success} = success$ and $E_{failure} = \emptyset$ (*i.e.* the error condition is the unreachability of the success event of the child).

- the `influenced_by` relation is used to express soft dependency: the rover motion and the UAV traversability mapping do not have a strong dependency, but the execution time and efficiency of the rover's navigation can be greatly improved thanks to the UAV traversability perception. Note that it is the basis for scheduling in GPGP/TAEMS [Lesser et al., 2004], which greatly influenced its introduction in our model.
  This relation has no requirements and no error conditions.
- the `planned_by` relation expresses that finding a way to execute an action is handled by a specific task. It can be used for all kinds of planning: for instance to represent the generation of task plans and the generation of paths (`PathPlanning` task in the rover plan). Child tasks in this relation can have two roles: (i) find a *first plan* and (ii) improve continuously this plan.
  For that reason, there is no requirement in the second case – *i.e.* if the parent task of the relation is not abstract. In the first case, the relation requirement is the emission of an event which indicates that the task has found a plan (`success` on one-shot tasks, another intermediate event on continuous planning tasks). There is no other error conditions than the unreachability of this event.

3.5 Error and error handling

In our system, errors can come from multiple sources:

- violations of the constraints defined by the task relations
- violations of constraints detected by external plan analysis tools. For instance, temporal constraints that are not described by the task relations.
- exceptions coming from the code itself: since the tasks are directly implemented using an all-purpose programming language, it is subject to the normal language exception handling mechanisms.

Regardless of the error source, the system describes errors by their type, and manage their types in hierarchies – much alike what is done for exception handling in modern programming languages. What is specific to our system is that errors may be tied to a particular plan object (task or event), thus describing what is the *source* of the error.

If that information is not available, no error handling is possible since the plan model does not provide enough information to represent what is the *consequence* of the error. If the error source is known, then three different error handling mechanisms are available. Two of these three mechanisms are already available in the literature but no system offer the three. We will discuss why they are complementary.

Immediate repairs an event handler, tied to the faulty event, transforms the plan so that, after the transformation, the event emission is not considered as an error anymore. For

instance, the event handler could restart the task that has faulted. The structure of the execution cycle (propagate then analyze) allows this behaviour.

Exception handling an error handling procedure which is able to match the specific error type is searched by going up in the `realized_by` graph. When a matching procedure is found, it is called and is supposed to repair the plan – for instance by replanning. PRS [Ingrand et al., 1996] has a very narrowed-down version of this (the parent procedure tries to find a new alternative plan on exception, and exceptions are not typed). TDL [Simmons and Apfelbaum, 1998] uses exception handling as present in modern programming languages, since the procedural executive has a tree structure. Concurrent Reactive Plans define adaptors which are akin to exception handlers as well. The specificity of adaptors is that they explicitly transform the plan, which could allow for instance to stop the current plan, perform some repairing actions and resume the normal course of actions.

This method represents the error repair mechanisms that are tied to the task hierarchy (high-level transformations for low-level faults). Its limitation is that it requires the repair to be done synchronously. Otherwise, a plan-based representation is required.

Plan repairs a task is tied to a possible fault, represented by an event. When the event is emitted *and* represents a fault (for instance, because it is forbidden by a relation), the repair task is started and its effect is expected to be the repair of the plan. When the task is running, the plan is left as-is, thus containing both the error and the repair – representing therefore the exact situation (the plan has failed *and* is being repaired). When the task is finished, the repair task is not taken into consideration anymore and if the error is not repaired yet, exception handling is used.

This method allows to represent error handling as an alternative execution path in the plan. Among other things, it allows to set error handling procedures that cross the boundaries of the plan managers since the repairing subplan can use all the multi-robot features of our plan manager. Finally, unlike the exception mechanism, it allows for non-synchronous repair of the plan: the repair task is a progressive task and can take quite some time.

3.6 Joint plans: ownership and roles

We need one more information to represent multi-robot plans: in a given plan, who is supposed to do what. To represent that, a task instance has an *ownership* attribute, which is the set of plan managers which are responsible for this activity. For local tasks, ownership is naturally set to the local plan manager only, for remote tasks it is set to the remote plan manager which is handling the task, and for joint tasks it is set to all the plan managers involved in the joint task.

This ownership attribute has two meanings: first, it describes the plan managers that are actually *executing* the task. Second, it describes the plan managers which are allowed to *modify* it: if it were possible to freely create relations between all plan objects, regardless of the plan managers which own them, then one plan manager would be able to constrain another into doing something. This should only be possible through negotiation: the golden rule of multi-robot plan management in our system is that a remote plan manager cannot change a robot plan without its consent. To ensure this decoupling, the following rules are in effect:

- to *remove a relation*, it is sufficient to only own one of the two objects involved. A plan manager can for instance freely remove a `realized_by` relation between one of its own task and a task owned by another plan manager (or a joint task). This is needed because one robot should be able to remove itself from a joint plan without negotiation, in cases of emergency for instance.
- any robot can remove itself at any time from the list of owners of a joint task. Ownership removal is then notified to the other plan managers.
- as for the removal of task relations, to *add or remove an event relation*, the local plan manager must be the owner of the child in the relation: it allows one plan manager to synchronize itself on events of another plan manager, which does not really affect the other plan manager.

Any other modification involving an object not exclusively owned by the local plan manager is not allowed in the executed plan. To handle the negotiation process needed for such modifications, we defined the *transactions*, which act as distributed whiteboard to change plans (see section 5).

One other attribute, related to multi-robot, exists in our system: the notion of role. This notion is now quite common in the domain of multi-agent and multi-robot systems. To quote Tambe [Tambe, 1997], who is at the origin of this notion:

> "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 directly integrate team management tools. In our plan manager, roles can be represented in two ways:

- an *explicit* mapping from each role to the plan manager fullfilling it
- by combining a structure pattern matching object and the ownership attribute: the corresponding role is fulfilled by the owner of the joint task which owns the tasks matching the given pattern. This allows to automatically update the role mapping when the plan structure is transformed.

¡¡¡¡¡¡¡ HEAD:main.tex ======= ¿¿¿¿¿¿¿ 86dd0dac22971a443f41129c4aa8099f33c64fc3:main.tex

3.7 Discussion

¡¡¡¡¡¡¡ HEAD:main.tex =======

¿¿¿¿¿¿¿ 86dd0dac22971a443f41129c4aa8099f33c64fc3:main.tex

Using events, our system can represent the execution flow in a way completely separate from the activities. This way, it is possible to define combination of events in a much richer way than in traditional supervision systems, in which the execution flow is built upon combinations of activities. It allows in particular to represent interacting progressive tasks, which is something other systems cannot do: for instance, one can easily represent the fact that, in our scenario, the data transfer should be triggered when the traversability map of the UAV is updated (see Fig. 8). Finally, we can represent situations which are defined outside a particular activity (for instance a `low_battery` event).

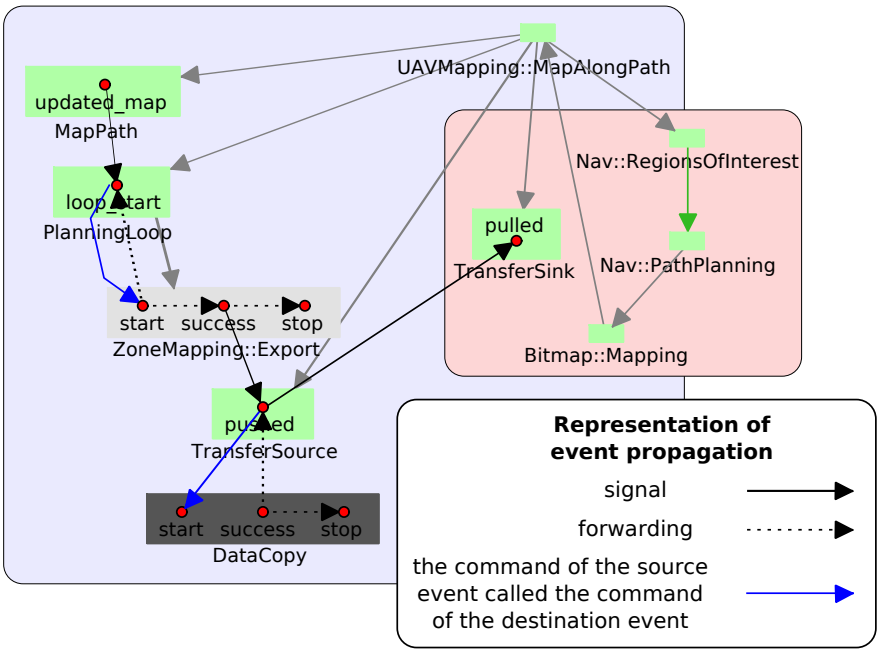


Figure 8: Data transfer after a map update on the UAV (experimental data). When the mapping is finished on the UAV, the rover is notified that new information is available. The data itself is passed through another channel established by the `TransferSink`/`TransferSource` pair. The rover integrates this new information and may update the list of regions of interest.

The second most important contribution of this plan model is the representation of joint plans (similar to GPGP/TAEMS) in a rich task-based formulation like Tambe's. This contribution is important since we want our system to act as an integration place for interaction schemes.

4 Plan Execution

In our system, the basis for multi-robot plan execution and multi-robot plan modification, is a distributed object management protocol which is not extensively presented in this paper. In this protocol, each plan manager knows, for each object (task or event) of its plan, what are the other representations of the same objects on remote plan managers. When this object is modified (through for instance event emission or relation modification), the plan manager notifies the involved remote plan managers of this modification.

This section focuses on problems related to multi-robot plan execution: how the execution of joint tasks in our system is related to the joint intention theory, how plan managers communicate with each other and finally which are the possible synchronization problems and how they are handled.

4.1 Joint tasks and joint intentions

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 manager 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 [Cohen et al., 1998]: 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 a `start` event is therefore equivalent to establishing that the started task should be realized as a joint persistent goal:

- all involved robots believe that the task is not yet achieved.
- they all accept to achieve the task (*i.e.* they *want* the task to be achieved).
- they continue until they reach 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 beliefs about the 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 `failed` event emission.

4.2 Communication management

Multi-robot systems cannot take communication for granted: interactions between plan managers can therefore not rely on a permanent network link. In our system, two interacting plan managers are *connected*. Connections are either *alive* if a communication link exists, or *dead* if there is no communication link.

In the plan, connections are represented by `ConnectionTask` instances, with remote tasks being dependent of this connection task. Therefore, if a separate component determines that we cannot rely on the remote robot, for instance because the connection has been dead for too long or the robot is late at a rendezvous, it simply stops the corresponding `ConnectionTask` and lets the usual plan manager recovery mechanisms apply (not described here, see [Joyeux et al., 2007] and [Joyeux, 2007]). These monitoring situations can also be expressed in the plan, for instance by making the `ConnectionTask realized_by` monitoring tasks. That way, `ConnectionTask` will be marked as `failed` if the monitoring fails.

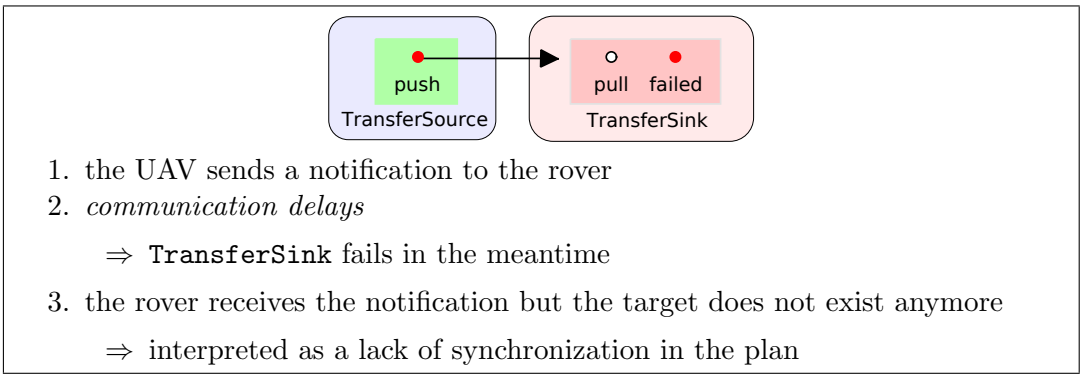


Figure 9: A situation in which a synchronization error during distributed plan execution is interpreted by our system as a plan fault. This plan fault can then be repaired by the usual means

Moreover, remote tasks are represented locally by task proxies, which are passive objects in general. Specific proxies can be defined for specific task models to predict the remote task state in a disconnected scenario, like the proxies of Machinetta [Schurr et al., 2005]. This allows to make the plan continue its execution without having a communication link. However, recovering after a bad prediction is an unresolved issue.

4.3 Synchronization issues during execution

For a single robot, our system has a synchronous event-based execution model. However, the synchronous properties of event propagation are obviously lost when executing multi-robot plans. This leads to interesting issues regarding synchronization during modification and execution of distributed plans.

The transaction tool which is presented later properly handles synchronization during plan modification. Synchronization for execution is less clear cut: each robot must be able to execute as much of its own plan as possible without constantly relying on communication with the other plan managers. This can lead to inconsistencies when a message, once processed, refers to plan objects which do not exist anymore. Our communication protocol detects such synchronization problems and usually interprets them as an error in the communication layer, which terminates the connection. Automatic plan cleanup mechanisms, not presented in this paper, then properly terminate the joint plan which was relying on the robot-to-robot connection.

However, one instance of this problem is instead interpreted as a lack of synchronization in the plan: the case of signalling/forwarding. Consider the situation represented in Fig. 9, where a signal message has been sent by the manager which owns the source event towards the manager owning the signal destination. In this case, a lack of synchronization generates a *plan error* and can be handled by the normal error reporting mechanisms of our plan manager.

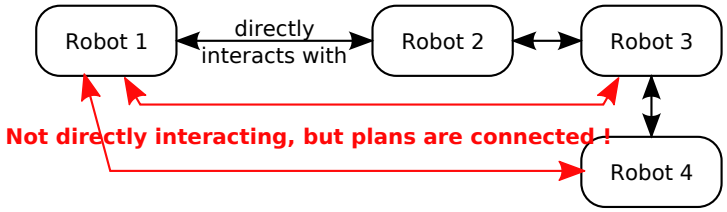
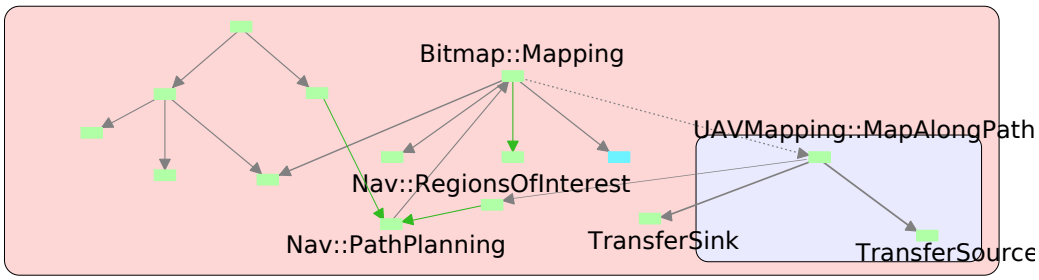
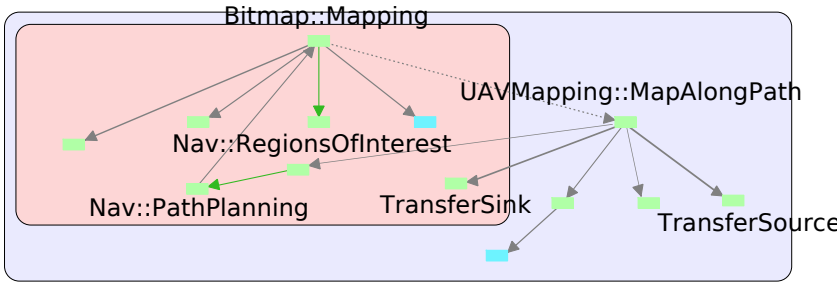


Figure 10: How a chaining effect can lead to arbitrary plan sizes in multi-robot context: in this schema, robots are only interacting with their direct neighbours. The whole system plan is therefore the union of all the robots plans, while the interactions are kept simple.



Joint plan as seen by the rover: approx. 65 tasks, 2 remote tasks. The `TransferSource` and `TransferSink` tasks are linked through event relations which are not shown here.



Joint plan as seen by the UAV: approx. 17 tasks, 8 remote tasks. The UAV explicitly subscribed to the rover's `Bitmap::Mapping` task, which is why the neighborhood of this task is also present (see text for details).

Figure 11: Joint plan as seen by the two robots, taken from experimental data. This figure illustrates how each robot has only a partial view of its peer's plan. The `TransferSource`/`TransferSink` pair is implementing the `DataTransfer` joint task presented in the introduction, as the management of joint tasks was not ready at the time of the experiment.

5 Plan Adaptation

5.1 Composition of partial plan views

In multi-robot system, it is impractible or even impossible to keep a full representation of the whole multi-robot plan: communication is limited and chaining effects (see Fig. 10) can indefinitely grow the size of the resulting plan[3]. Two mechanisms exist for handling, in each plan manager, the partial view of the other robots' plans.

The first one is an automatic scheme which maintains the minimal set of remote tasks which must be represented for the distributed plan management to work properly. In our system, this is the set of plan objects which are directly related to the robot's own objects through a task or event relation (Fig. 11): they are directly interacting with the robot's own plan.

The second one is a manual mechanism in which a plan manager subscribes to another's task or event. In that case, the subscribed plan manager will be notified of all changes regarding the plan object, including changes in its neighborhood (*i.e.* relations in which the subscribed object is involved, see `Bitmap::Mapping` in the UAV plan on Fig. 11). One of the roles of distributed plan building is to determine what tasks, outside of the automatically subscribed ones, are relevant to the remote plan managers.

5.2 Plan-based negotiation

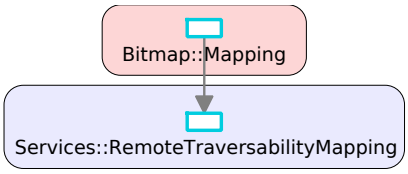
In a mono-robot plan manager, a transaction is a whiteboard used to build a set of plan modifications without modifying the plan being executed. The plan manager can then synchronously apply this changeset to the executed plan or discard it. It ensures that the executed plan is always sound, provided that the planners themselves are producing sound changesets.

In a multi-robot context, transactions are shared among plan managers, and they can change multiple plans at the same time. They can therefore be used as a basis for negotiation (Fig. 12): one robot builds a partial multi-robot transaction, which can then be modified by others, until an agreement has been found on the new joint plan, in which case the transaction is *committed* into all the involved plans. The transaction is a sandbox in which every kind of plan change can be proposed. Typical modifications are: task and event relations, ownership, roles, and subscription.

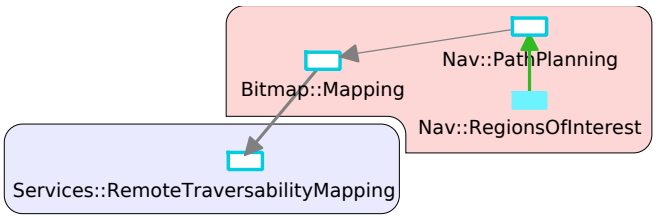
More formally, a transaction is defined by a tuple $(P, O_{new}, O_{removed}, O_{proxies}, R)$, where:

- P is the plan the transaction applies on. It can either be the executed plan or another transaction.
- O_{new} is a set of tasks and events that are not in P.
- $O_{removed}$ 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 mechanism: the tasks that 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. That transaction-specific representation also allows to change task attributes like ownership, roles, ...

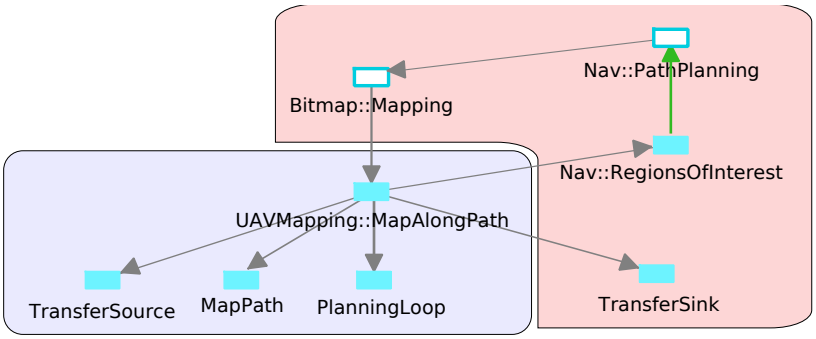
[3]The argument for full plan representation we made in the introduction holds for single robots only, where plan size remains limited.



Abstract proposal: the UAV builds a transaction to *propose* the interaction to the rover. For now, the UAV did not choose the interaction modality it will use.



The rover accepts the transaction: the rover receives the transaction and accepts the proposed interaction. It adds a new `RegionsOfInterest`, not yet present in its plan, to propose the corresponding service to the rover.



Final state of the transaction: the UAV decides to use the proposed `RegionsOfInterest` task and chooses the corresponding mapping modality (`MapAlongPath` task). It then adds the tasks needed by that modality and for the data transfer.

Figure 12: Negotiation steps between the UAV and the rover using a transaction (from experimental data). The blue-and-white tasks are representation of tasks which are already present in the plan and are modified by the transaction.

- 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.

The transaction therefore defines the relations (R) between a set of objects in P and new objects. Based on this information, the plan manager is able to determine if the plan changes inferred by the execution violates the constraints defined in the transaction and – through the decision control component – can initiate an adaptation step with the planner. For more information on this mechanism, please refer to [Joyeux, 2007].

During the commit, the plan managers ensure that the changeset contained in the transaction is either applied at the same time on all involved plans or not at all. A robot can therefore assume that, if new multi-robot relations have been added to its plan, the same relations have been added on the other plan managers as well. Accepting a transaction can therefore be interpreted as a form of *weak contract*: the contract represented by the plan in the transaction is accepted at the time of the commit, but can be broken later, either through re-negotiation, or because the evolution of the situation demands it.

Finally, the representation of a transaction as a plan transformation allows to make sure that, because the robots continued their execution, the plan included in the transaction does not become incoherent with the executed plan. A mechanism embedded in our system detects such cases of execution/planning conflicts and forbids the transaction commit if one appears. A negotiation protocol between the plan manager, the decision control component and the planning tools then allow to solve these conflicts if possible.

6 Implementation and Results

The plan manager described in this paper is currently implemented in the Ruby programming language, is Free Software and is available at `http://roby.rubyforge.org`. We use the object-oriented capabilities of Ruby as a way to define task models and task instances as classes and objects. In distributed contexts, Ruby allows to create classes on-the-fly, which allows to map unknown remote models to anonymous local classes. Moreover, developing the system in a general-purpose language promotes code reuse in supervisors. The development of our controllers shows that a great level of modularity can be achieved by defining mixins for patterns in task behaviours on the one hand, and libraries of often-used plan modification operators on the other hand.

6.1 Robot-specific controllers

We use two robots for our experiments: one iRobot ATRV owned by the LAAS, and one Yamaha RMAX helicopter owned by the french ONERA laboratory. Two controllers, one for each, have been implemented.

In the case of the rover, the Roby controller controls an already existing set of Genom modules that implement the robots' functional layer [Fleury et al., 1997]. Pocosim [Joyeux et al., 2005], our simulation system, allowed us to use most of the modules in simulation without modification. We only replaced the image acquisition chain by a simulation version of the elevation mapping module, which reads a pre-computed traversability map. This module offer the same interface as the real one: we keep the simulated functional layer as close as possible to the real one, to use the same controller in simulated and real environments. However, since no noise is added to perception, the functional layer output is much better than in real conditions.

On the UAV, the Roby controller is interfaced with the helicopter functional server through a network socket and a custom protocol. In simulation, the functional server is replaced by a compatible simulation-only server which reads pre-computed traversability maps.

In this implementation, we take into account the fact that the UAV can give traversability information of different quality, based on its perception altitude. The initial terrain and the apriori traversability maps used to simulate the UAV perception are shown on Fig. 13. The UAV maps are generated based on the elevation data using 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.

6.2 Plan execution

The joint plans built during negotiation include all the information required to manage its execution. The plan manager offers two error recovery mechanisms: either errors are handled in the plan (conditional plans), or an exception mechanism (not described here) is used. In multi-robot contexts, exception handling is not distributed: it is supposed to be a synchronous operation, and thus cannot be done in the asynchronous communication scheme we use. If multi-robot error recovery is needed, it shall be either directly expressed in the plan or done reactively by negotiation between the involved decision control components.

During execution, the UAV is therefore kept informed of any update of the rover's path. It can use this information to build and update its own mapping path and send the traversability updates to the rover. The rover then replans its path, sends the updated path to the UAV, and so on.

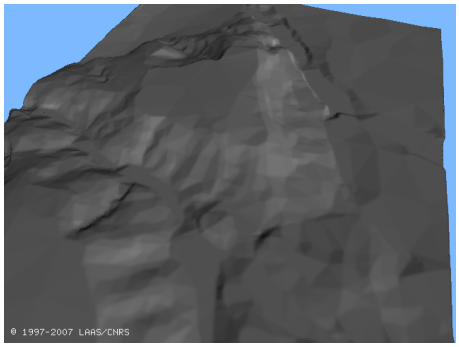
The interaction finishes either when (*i*) the influenced tasks (parent tasks in the `influenced_by` relation) have successfully finished, which would announce the success of the joint plan, or (*ii*) when the plan structure from which the systems initiated the interaction has ceased to exist. In that case, the rover's plan manager will notify the UAV of this change, and the UAV can then decide to change its plan accordingly. Note that this is not an automatic process of the plan manager, but a decision to be taken by the UAV.

6.3 Experimental results

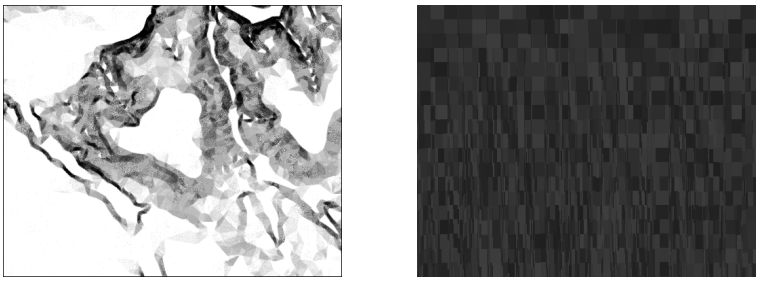
We used two scenarios to evaluate the plan managers: the rover alone, and rover/UAV cooperation with the UAV as a remote sensor. The single-robot case works as expected in both simulation and real world and is not the focus of this paper.

In simulation, the whole rover/UAV navigation takes around 30 minutes for a 400 meters navigation, at maximum 1 m/s speed for the rover. Key situations of the runs are shown in Fig. 14. The generated plans are around 65 tasks for the rover and 17 for the UAV. Our execution system is based on a fixed-length execution cycle, which in this scenario implementation has been fixed at 100 milliseconds, a speed which is easily reached by our implementation. However, latency issues still exist due to problems with the robot OS (which is not realtime), and with the garbage collector of the Ruby interpreter. Some key statistics of plan execution are summarized on Table 1.

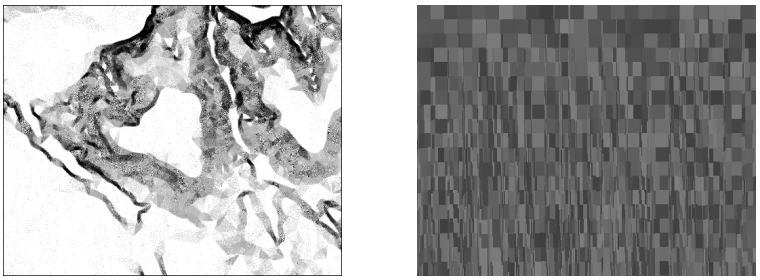
Real-world cooperation tests have partially validated the approach: the plan managers work as expected, but implementation issues in the autonomous rover navigation did not allow to finish the whole scenario.



Elevation map of the terrain used in simulation



Traversability and confidence maps for simulated high altitude perception

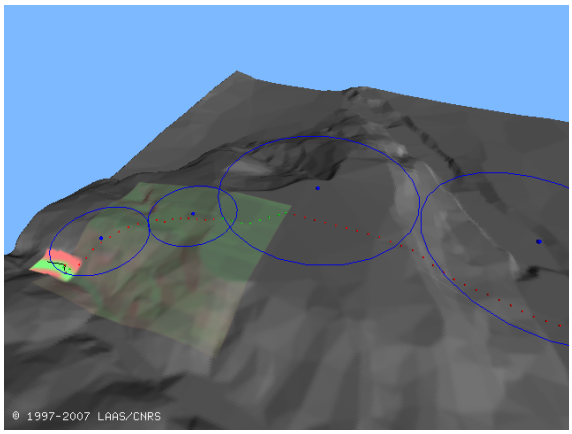


Traversability and confidence maps for simulated low altitude perception

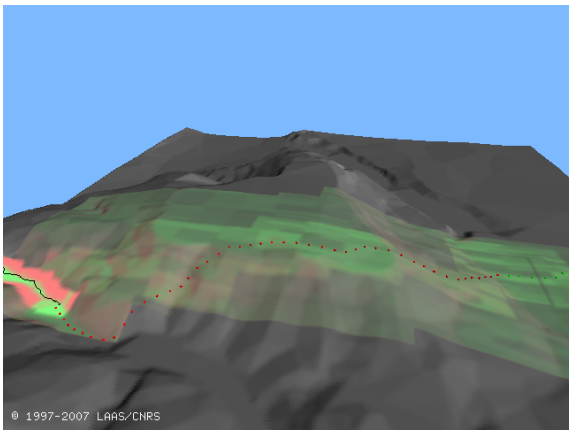
Figure 13: Maps used for the simulation of our rover/UAV scenario. The confidence maps are random maps generated so that they form small "patches" of constant confidence. This resembles the data obtained in the real setup, which uses a texture-based algorithm..

	total	max p. cycle
event emissions	23490	24
event command calls	11518	15
transaction commits	1180	3

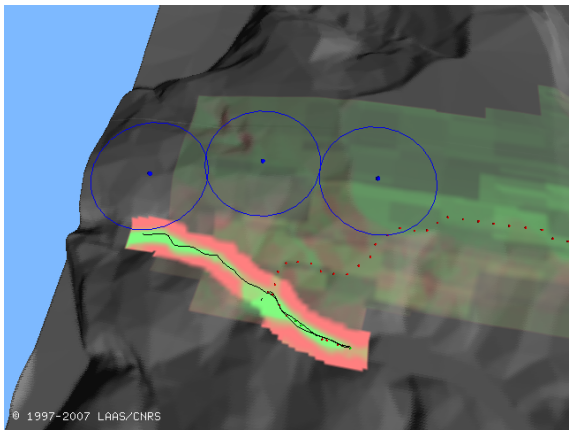
Table 1: Execution statistics during the execution of the rover/UAV scenario.



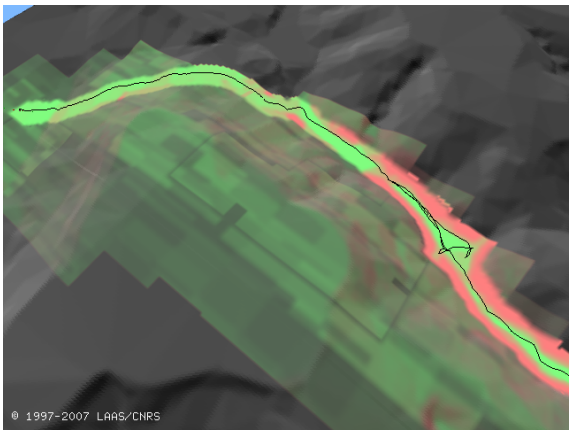
(i) the UAV perceived the first zone for the rover. Low-altitude goals (blue circles) are generated in this newly perceived zone. High altitude goals remain for the terrain still unknown



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



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



(iv) final situation

Figure 14: 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.

6.4 From the experiment, back to the implementation

The implementation of both controllers and the cooperation scenario allowed to test the basic concepts of the plan manager.

Transactions All plan generation in the rover is done asynchronously in order to test the concept of transactions. Moreover, the use of distributed transactions and triggers for the rover/UAV cooperation is 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 is a great asset for the development of our controllers: the system can represent and safely handle problems during development, which particularly assists debugging the functional layer.

Extensibility Building more complex objects on top of the basic system presented in this paper is quite easy. The 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.

7 Conclusion and Outlook

In this paper, we have presented *multi-robot* aspects of our *plan manager* component. Its single-robot features, such as exception handling or details of the event and transaction mechanisms, are presented in prior publication [Joyeux, 2007, Joyeux et al., 2007]. The underlying *plan model* separates task representation from execution, which provides (i) flexibility for design, implementation, and system integration; and (ii) allows to represent and execute joint plans in multi-robot context. In particular, it allows to represent interactions between progressive tasks such as the mapping processes described in the example.

Based on this model, the main contribution is a *generic* and *distributed* plan manager tool. It is generic with respect to the planners and the team management schemes used on each robot and in the multi-robot system, thus providing a powerful tool for system integration. The main advantage of distribution lies in the fact that it can handle non-permanent communication while simultaneously executing the plan on each robot. Within the plan manager, the *transaction* mechanism allows to build plans cooperatively, negotiate plan modification, and express commitment of individual robots to the joint tasks.

Note that the plan model can express polymorphic task hierarchies, which allows us to represent other robot's tasks at an appropriate level of abstraction without sacrificing the coherence of the overall plan. Thus, the full multi-robot plan is represented, with varying degrees of simplification depending on the requirements of each joint task. We also presented a trigger mechanism as an expressive way to define how a robot can take advantage of the plan modifications that occur in its peers.

One can notice that the illustrating scenario used in this paper a bi-robot scenario, not a multi-robot one. This is due to the current state of our implementation, which lacks robustness for full-blown multi-robot interaction. However, the principles presented in this paper are in no way limited to the bi-robot case:

- our whole model and execution engine is based on the representation of activities that are running in parallel. This is a natural representation for the activity set of a multi-robot systems, and as such the mechanisms built on top of it will also work for multi-robot systems.

- role-based multi-robot task management is shown to be an efficient representation of the place of each agents in the plan of themulti-robot system.

- from a plan-building point of view, the design has been driven by the necessity to integrate multiple decision-making processes in a single robot ([Joyeux, 2007]). The mechanisms put into place to handle this will naturally also handle the multi-robot case, thanks to the distributed transaction tool.

However, the *scalability* question remains: how well will scale the communication mechanisms needed to propagate events in networks of robots and to propagate plan changes.

Future work will address extensions to the model and the manager, as well as extended usage for system integration in other projects. The implementation described in this paper has shown promising results for the integration of a modular functional layer like Genom in a plan management system, and for the development of interaction schemes. We expect to use it in further projects to design and integrate interaction schemes, planning tools and functional layers.

Concerning the plan model, the inclusion of temporal information will allow us to better prepare for communication loss, for example by setting up rendezvous points. We expect the event model used in our system to support this quite naturally: one can embed a time propagation algorithm based on the event graphs and on temporal prediction provided by the tasks.

On the level of plan management, we will work on defining and implementing negotiation protocols between robots as well as between different planners on each robot. The scenario presented in this paper remains a good testbed for such endeavour: the robustness of the team could be greatly improved by explicitly handling communication loss during planning, and by taking into account the UAV's planned perception during the planning of the rover's navigation.

References

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

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

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

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

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

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

[Bresina et al., 2005] Bresina, J. L., Jonsson, A. K., Morris, P. H., and Rajan, K. (2005). Mixed-initiative planning in MAPGEN: Capabilities and shortcomings. In *Proceedings of the ICAPS Workshop on Mixed-Initiative Planning and Scheduling*.

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

[Clement and Durfee, 1999a] Clement, B. J. and Durfee, E. H. (1999a). Indentifying and resolving conflicts among agents with hierarchical plans. In *Proceedings of the AAAI Workshop on Negotiation*.

[Clement and Durfee, 1999b] Clement, B. J. and Durfee, E. H. (1999b). Top-down search for coordinating the hierarchical plans of multiple agents. In *Proceedings of the third annual conference on Autonomous Agents*.

[Cohen et al., 1998] Cohen, P., Levesque, H., and Smith, I. (1998). On team formation. *Contemporary Action Theory*.

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

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

[Gancet et al., 2005] Gancet, J., Hattenberger, G., Alami, R., and Lacroix, S. (2005). Task planning and contrl for a multi-uav system: architecture and algorithms. In *Proceedings of IEEE IROS*.

[Gancet and Lacroix, 2003] Gancet, J. and Lacroix, S. (2003). 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)*.

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

[Joyeux, 2007] Joyeux, S. (2007). *A Software Framework for Plan Management and Execution in Robotics: Application to Multi-Robot Systems*. PhD thesis, ISAE. `http://tel.archives-ouvertes.fr/tel-00283086/fr/`.

[Joyeux et al., 2007] Joyeux, S., Lacroix, S., and Alami, R. (2007). A software component for simultaneous plan execution and adaptation. In *Proceedings of the IEEE IROS*.

[Joyeux et al., 2005] Joyeux, S., Lampe, A., Lacroix, S., and Alami, R. (2005). Simulation in the LAAS architecture. In *Workshop in Software development in robotics, ICRA 2005*.

[Lesser et al., 2004] Lesser, V., Decker, K., Wagner, T., Carver, N., Garvey, A., Horling, B., Neiman, D., Podorozhny, R., NagendraPrasad, M., Raja, A., Vincent, R., Xuan, P., and Zhang, X. (2004). Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 9(1):87–143.

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

[Myers, 1996] Myers, K. L. (1996). Advisable planning systems. In Tate, A., editor, *Advanced Planning Technology*. AAAI Press, Menlo Park, CA.

[Myers, 1998] Myers, K. L. (1998). Towards a framework for continuous planning and execution. In *Proceedings of the AAAI Fall Symposium on Distributed Continual Planning*.

[Myers et al., 2003] Myers, K. L., Jarvis, P. A., Tyson, W. M., and Wolverton, M. J. (2003). A mixed-initiative framework for robust plan sketching. In *Proceedings of the 2003 International Conference on Automated Planning and Scheduling*.

[Pollack and Horty, 1999] Pollack, M. E. and Horty, J. F. (1999). There's more to life than making plans: Plan management in dynamic, multiagent environments. *AI Magazine*.

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

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

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

[Smith, 1980] Smith, R. G. (1980). 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.

[Tambe, 1996] Tambe, M. (1996). Agent architectures for flexible, practical teamwork. In Senator, T. and Buchanan, B., 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. American Association For Artificial Intelligence, AAAI Press.

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

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

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