A plan manager for multi-robot systems

Sylvain Joyeux[1], Rachid Alami[1], and Simon Lacroix[1]

LAAS-CNRS Université de Toulouse firstname.lastname@laas.fr

Summary. This paper presents a software component, the plan database, which provides the services needed to build and execute plans in a multi-robot context. This plan database 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 simulated example that involves a rover and an UAV in an initially unmapped environment.

1 Introduction

One can distinguish two main schemes in today's distributed robotics architectures. On the one hand, some approaches rely on a team-management layer which sends orders to a mono-robot supervision system (for instance Teamwork [9] and TraderBots [1]). In these approaches, interactions between robots are reactively managed, and are not pro-actively managed within a plan structure. Systems based on such a scheme cannot predict that, for instance, interactions will be needed in the future. Also, the team-management layer is limited to managing high-level tasks, which makes difficult to handle tighter interaction modalities, like opportunistic cooperation, which take place at a lower abstraction level.

On the other hand, some approaches rely in mono-robot supervision systems that can send orders transparently to other robots: in FIRE [8], a robot can change the state of a remote robot, but not pro-actively, and it assumes that communication is available at all times, which can not be taken for granted in multi-robot systems. Coupling such supervision layers with a team-management component gives better result (like FIRE, which is a modified Task Desciption Language coupled with a Contract-Net Protocol task allocation scheme), but the resulting systems still have the problem of the above 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 ac-

count the interaction in the lowest one. They lack a common place in which to describe all the interactions in place.

To our knowledge, three architecture have implemented plan-based cooperation: GPGP and its associated task representation TAEMS [6], the COMETS [3] architecture and Machinetta [7]. 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 an 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. Machinetta defines a plan-based interaction mechanism and an architecture for the interaction. The main focus of COMETS and Machinetta is to define a generic team management system and the one of GPGP is multi-robot planning. All three 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.

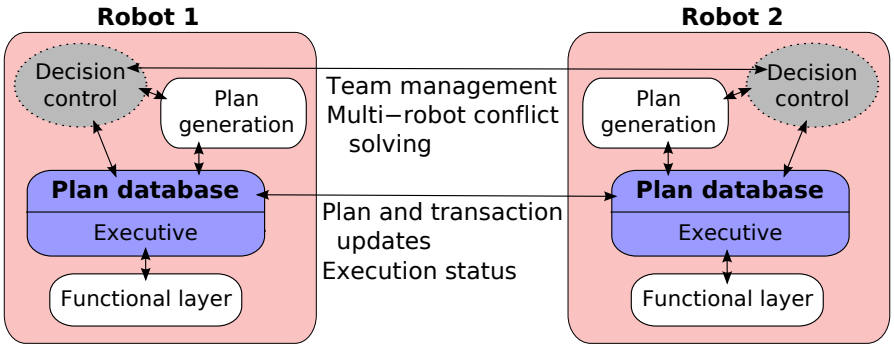


Fig. 1. In this architecture, each robot plan is managed by its own plan database. Parts of these plans are shared among databases. Decision control manages plan generation, team management and solves conflicts between execution and planning

In this paper, we present a plan database component, whose role is to provide the services required to manage and execute multi-robot plans. The component is generically designed, and is not associated to specific plan generation systems or specific team management schemes. This plan database acts as the middle-man between the functional layer and plan generation components (Fig. 1). In this architecture, the planners are responsible for producing coherent plans either for the robot alone or for the team, and the functional layer is a service layer which provides the perception and action algorithms that interfaces the robot and the real world. Between these two, we introduce three components: the *plan database* maintains a plan, which is a graph of

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

tasks and events defining what the robot may do in the future and how it will do it. This plan is continuously interpreted by the *executive* to produce actual actions, using an event-based model. 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 possibilities with other robots. Second, it handles the choices that have to be made during execution: since our framework allows simultaneous planning and execution, conflicts will arise between the main plan being executed 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.

The plan database (pDB) has been designed with the following goals:

- provide the tools required to build multi-robot plans, and use plans as a basis of negotiation. In a pDB, 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.
- manage plan modifications atomically: the executive must not execute partial plans that are being generated by planners, as they are not sound yet.
- express multi-robot execution contexts, and handle them during execution. In particular, plan execution is able to handle the fact that robots cannot communicate at all times.
- allow integration of external tools to handle multi-robot decision processes like task allocation and multi-robot planning.

2 Plan-based cooperation in a rover/UAV scenario

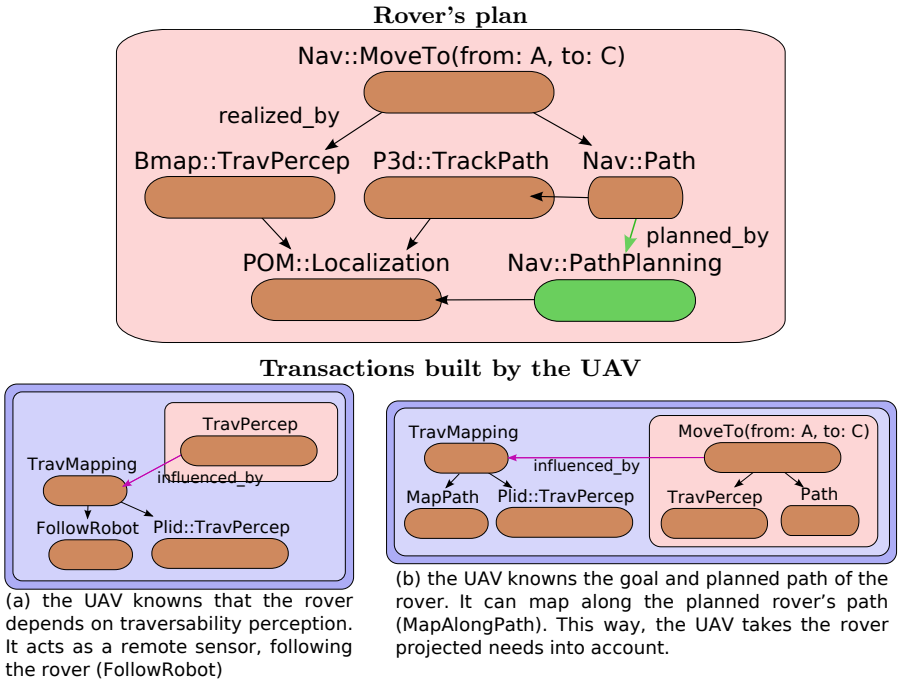
As a supporting example to present the concepts associated to the pDB, we outline here a scenario which has been partially realized in simulation[2]. In this scenario, the mission to achieve is a rover `GoTo` task in an initially unknown terrain. For that purpose, the rover is endowed with algorithms which instance a perception/decision/action loop, summarized by the three tasks `Bmap::TravPercep`, `Nav::Path` and `P3d::TrackPath`. These tasks respectively build a traversability map from perceived data, plan a path in the traversability map and execute the planned path. The rover is assisted by an UAV, which is able to update the traversability information of the rover thanks to a vision process (`Plid::TravPercep`). 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 simple scenario, we will show in this section how the rover-UAV cooperation can be expressed as relations between individual robot plans, and how this joint plan is managed in our plan database.

Fig. 2 presents the initial rover plan, in which `P3d::TrackPath` follows the waypoint list, `Nav::Path`, established by `Nav::PathPlanning` on the ba-

[2] Its realization with real robots is currently under way

sis of the results of `Bmap::TravPercep`[3]. The arrows between tasks express relationships between tasks, which are described in section 3.1.



(a) the UAV knowns that the rover depends on traversability perception. It acts as a remote sensor, following the rover (FollowRobot)

(b) the UAV knowns the goal and planned path of the rover. It can map along the planned rover's path (MapAlongPath). This way, the UAV takes the rover projected needs into account.

Fig. 2. *(top)* partial view of the initial rover plan. *(bottom)* two alternative cooperation scheme, based on two different patterns found in the rover's plan, in transactions built by the UAV.

We selected for this scenario an opportunistic method to initialize the interaction: the rover does not know beforehand that an UAV will help to realize its mission. When the UAV detects the rover presence by using an automatic network discovery mechanism, it will add *triggers* on the rover's plan. These triggers are used to match specific patterns in plans, and a list of known patterns are defined in each robot controller. The UAV will therefore be notified of a pattern it is able to interpret.

The two alternatives presented in Fig. 2 are based on two different triggers. When multiple triggers match the same task pattern, the choice of the modality is not a decision made by the pDB, but by the decision control. The pDB only notifies decision control what task set triggered what pattern. To achieve this choice, decision control can for instance build multiple transactions with the multiple possible outcomes and compare the resulting plans.

Let's assume that the (b) modality is chosen by decision control. The UAV then uses a planner to produce the plan, based on its partial knowledge of the

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

rover's plan. This planner cannot directly change the current rover plan: first, partial plans are not directly executable, and second, if the UAV had such an ability, it would mean that the rover is not able to fully control its activities. We therefore need a way to change plans outside the plan being executed, so that *(i)* the executive only sees complete executable plans and *(ii)* one robot can propose a plan modification to another. This is achieved by *transactions*, that represent a plan changeset, *i.e.* set of modifications required to get from the current main plan to the desired final plan. Transactions can be shared and synchronized across pDBs.

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 main plan accordingly at the same time, and can start its execution. If a joint plan cannot be found, the transaction is simply discarded.

3 A distributed plan database

The plan database has been designed as a tool for plan management, in both mono-robot and multi-robot systems. In a multi-robot context, a single pDB 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 that 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 pDB is informed only about the remote tasks it is interacting with (section 3.3). This allows to keep each pDB plan at a tractable size regardless of the number of pDB currently interacting.

The section first describes the pDB model of tasks and events, and how these objects interact which each other. It then presents how multi-robot plans are managed and built, and finally the way to handle situations where two interacting pDBs do not have communication, which is a key feature in a multi-robot system.

3.1 Plan model

We summarize here the plan model used in the pDB – more details on this model, and the general plan execution mechanisms and on error recovery are detailed in [4].

In the pDB, plans are graphs of two kind of objects: events and tasks. Events describe *what* occurs during task execution (for instance a `stopped` event is emitted when a task has stopped), and *the way* they occur: an interruptible task has a *controllable* `stopped` event, which means that the controller can call the event command and make the task stop. The main event relation is the *signal* relation: if an event e_1 signals a controlable event e_2, then

e_2's command will be called when e_1 is emitted. The signal graph therefore describes the action(s) to take when an event is emitted.

Similarly to TDL and other task-based systems, tasks describe processes, and the task relations describes the interactions between these processes.

Tasks models are defined as a set of events, the basic ones being `started`, `failed`, `success` and `stopped`, which define milestones in the task execution. These task models are put into a hierarchy in which the more abstract task models are refined in less abstract ones. In the plan, these models are used to define task *instances*, which are the *active* objects: they determine when the events should be emitted, handle event commands, and can locally repair the plan in case of exceptions. Defining a task model hierarchy has the advantage that, in multi-robot contexts, a robot does not necessarily have to know all the models known by all the pDBs. Instead, it can use an intermediate level of abstraction to represent another's specific task implementation (compare the specific models of the rover and the more generic ones in the UAV transactions on Fig. 2).

Three task relations are currently defined:

- The `realized_by` relation is used to express dependency: the successful execution of `Nav::MoveTo` requires its two child tasks to be successfully achieved.
- The `planned_by` relation expresses that finding a way to execute an action is handled by a specific task.
- The `influenced_by` relation: the rover motion and the UAV traversability mapping do not have a strong dependency, but the execution time and efficiency of the motion can be greatly improved thanks to the UAV's `Plid::TravPercep` task.

3.2 Multi-robot plans

In order to represent what tasks are executed by whom, a task instance has an *ownership* attribute, which is set to the list of pDBs which are executing this task. For local tasks, ownership is naturally set to the local pDB only, for remote tasks to the remote pDB which is handling the task, and for joint tasks to all the pDBs involved in the joint task.

During execution, a pDB is transparently notified of all changes that are done to remote tasks that are present in its own main plan, in a manner robust to communication failures (section 3.4). This includes execution status (signalling, event emission), and changes in the event and task graphs, which allows the distributed execution and adaptation of multi-robot plans. However, we can easily see that if it were possible to freely create relations between local and remote objects, then one pDB would be able to take decisions for another, which should only be possible through negotiation: the golden rule of multi-robot plan management in our system is that a remote pDB cannot change a robot plan without its consent. To ensure that, the following rules are in effect:

- to *remove a relation*, it is sufficient to only own one of the two objects involved. A pDB can for instance freely remove a `realized_by` relation between one of its own task and a task owned by another pDB (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 pDBs.
- as for the removal of task relations, to *add or remove an event relation*, the local pDB must be the owner of the child in the relation: it allows one pDB to synchronize itself on events of another pDB, which does not really affect the other pDB.

Any other modification involving an object not exclusively owned by the local pDB is not allowed in the main plan. To handle the negotiation process needed for these modifications, we added *distributed transactions*, which act as distributed whiteboard to change plans.

3.3 Building multi-robot plans

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

In a multi-robot context, transactions are shared among pDBs, and they can change multiple plans at the same time. They can therefore be used as a basis for negotiation: 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 then pDB ensures that the changeset contained in the transaction is 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 pDBs as well.

The update mechanism mentioned earlier uses subscription: a pDB subscribes to remote tasks to get all updates about them. Obviously, however, it must be automatically subscribed to remote tasks that are related to local tasks by the object graphs. One of the role of plan building through distributed transactions is to determine what tasks, outside of the automatically subscribed ones, are relevant to the remote pDBs.

3.4 Connection management

Multi-robot systems cannot take communication for granted: interactions between plan databases can therefore not rely on a permanent network link. In our system, two interacting pDBs 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 pDB recovery mechanisms apply (not described here, see [4]). 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.

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 [7]. This allows to make the plan continue its execution without having a communication link. Recovering after a bad prediction is however an issue with which our system is not dealing.

4 Implementation and simulation results

The pDB is currently implemented[4] in the Ruby language. 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.

4.1 Simulation setup

Two controllers, one for the rover and one for the UAV have been implemented, using an already existing set of Genom [2] modules that implement the robots' functional layer. Pocosim [5], our simulation system, allowed us to use most of the modules in simulation without modification. We only replaced the image acquisition chain by modules (`plid` on the UAV and `bitmap` on the rover) which read a pre-computed traversability map (Fig. 3). These modules offer the same interface as the real ones: we keep the simulated functional layer as close as possible to the real one, to use the same controller in simulated and real environments. Since no noise is added to perception, the functional layer output is much better than in real conditions.

4.2 Plan execution

The joint plans built during negotiation include all the information required to manage its execution. The pDB offers two error recovery mechanisms: either

[4] available at `http://www.laas.fr/~sjoyeux/research.php`

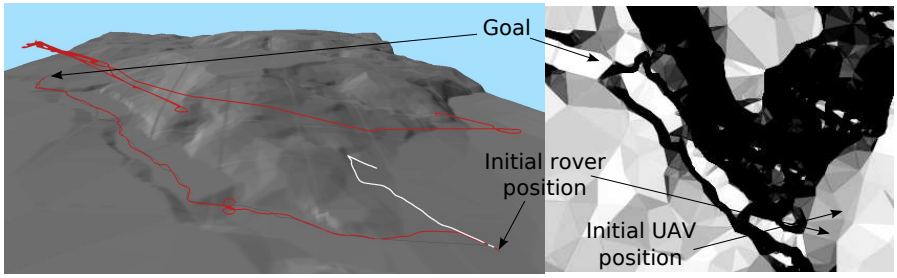


Fig. 3. (*left*) rover trajectories with (red) and without (white) the UAV's help, and the UAV trajectory in map-along-path mode. (*right*) Traversability map, white being perfectly traversable and black not traversable. This map is discovered incrementally by the rover and the UAV through the `Bmap::TravPercep` and `Plid::TravPercep` tasks.

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 negociation between the involved decision control components.

During execution, the UAV is therefore kept informed of any update of the rover's path (`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 pDB will notify the UAV of this change, and the UAV can then decide to change its plan accordingly. Note that it is not an automatic process of the pDB plan management, but a decision to be taken by the UAV.

4.3 Simulation results

We have run three different kinds of simulations: the rover alone, with the UAV as a remote sensor, and with the UAV which maps along the rover path (Fig. 3 displays the resulting trajectories). In these simulations, the rover alone don't always reach its goal. The two interaction schemes are consistently successful. The rover reaches its goal (at 200 metres) in 10 minutes. The two setups are equivalent in this simulation because the rover quickly gets trapped in the corridor we see in the terrain.

5 Conclusion

The contributions of this paper are the multi-robot extension to a plan model, and the set of tools embedded in the plan database to build and execute multi-robot plans. The pDB transactions allow to cooperatively build multi-robot plans, negotiate on them and commit them. Its distribution mechanisms allow to manage the plan during its distributed execution, including situations where communication is not permanent. Finally, by not relying on particular interaction schemes, the pDB is generic enough to allow integration of most of the existing schemes.

The framework 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. In the future, it should act in various projects as an integration tool for interaction schemes, planning tools and functional layers. Our main objective is now to integrate temporal information in the plans, which is required to properly handle communication loss for instance by handling the setup of rendezvous points.

References

1. B. Dias. *TraderBots: A New Paradigm For Robust and Efficient Multirobot Coordination in Dynamic Environments*. PhD thesis, 2004.
2. 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.
3. 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.
4. S. Joyeux, S. Lacroix, and R. Alami. A framework for simultaneous plan execution and adaptation. In *Proceedings of the IAV IFAC Symposium*, 2007. to be published.
5. S. Joyeux, A. Lampe, S. Lacroix, and R. Alami. Simulation in the LAAS architecture. In *Workshop in Software development in robotics, ICRA*, 2005.
6. V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. NagendraPrasad, A. Raja, R. Vincent, P. Xuan, and X. Zhang. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 9(1):87–143, July 2004.
7. N. Schurr, S. Okamoto, R. T. Maheswaran, P. Scerri, and M. Tambe. Evolution of a teamwork model. *Cognitive Modeling and Multi-Agent Interactions*, 2005.
8. 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*, 2002.
9. M. Tambe. Agent architectures for flexible, practical teamwork. In *Proceedings of the 14th National Conference On AI and the 9th Innovative Applications of AI Conference*. AAAI, 1996.