A FRAMEWORK FOR SIMULTANEOUS PLAN EXECUTION AND ADAPTATION

Sylvain Joyeux * **Rachid Alami** * **Simon Lacroix** *

* *LAAS/CNRS, 30 av. du Colonel Roche, 31000 Toulouse Cedex*
firstname.lastname@laas.fr

Abstract: This paper presents a software component, the plan database, which provides the needed services to define plans, execute them and more importantly adapt them during execution. This plan database handles fully dynamic plans (insertion and removal of tasks), defines task transformation operators and provides tools for safe concurrent execution and modification of plans. These features are essential in multi-robot and human-robot contexts, where tasks need to be easily passed between systems and plan adaptation helps coping with the unpredictability inherent to systems where multiple agent make decisions.

Keywords: architecture

1. INTRODUCTION

In robotic systems, the planner rarely produces results directly executable by the platform. It is often needed to have an intermediate component, whose role is to manage the functional layer, based on the execution state and the information available in the plan. Moreover, this execution component can handle "simple cases", so that upper layers do not have to manage all events that come from the functional layer: it typically performs error recovery and a limited form of script-based plan generation. To ease the development of these executives, tools like TDL (Simmons and Apfelbaum, 1998), OpenPRS (Ingrand *et al.*, 1996) or ESL (Gat, 1997) have been designed.

One problem with this approach is that two components, the planner and the executive, are keeping two different plans. The first often lacks information about the execution state, while the second lacks information about high level goals, which are usually handled by higher layers. One can therefore not manage the plan globally: an error in the lower layers cannot be linked to the upper-level plan, state and time estimation in the higher layers cannot use low-level knowledge.

On the contrary, in the Claraty (Estlin *et al.*, 2001) architecture, a single plan is being managed by a central component. Claraty then provides a simple mechanism to ensure that the part of the plan being executed will not be changed by the planner: a floating "line" separates the long-term plan, which can be freely modified by the planner, from the short-term which is handled by the executive and read-only for the planner.

The IDEA (Finzi *et al.*, 2004) architecture defines a hierarchy of agents, each of which plans a subset of the whole plan and then sends part of it to other agents for further processing. This allows for instance to use a fast, reactive planner for low-level tasks and a long-term planner for high-level ones. Since the IDEA planners share plan representation, they can maintain coherency in plans of all agents. The common plan model of all IDEA agents is however too restrictive to allow the use of many different planners.

All these architectures lack the ability to efficiently remove tasks from plans. The reason is often that the planners themselves lack this ability. Our experience in developping robot systems show that this capability is nonetheless essential when

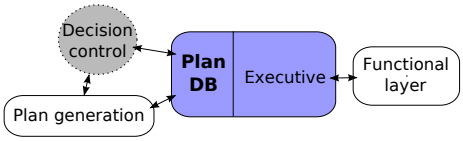


Fig. 1. The plan database is a system component that represents and maintains plans during execution

the mission set is highly dynamic. This paper outlines a plan management component, the plan database (pDB), which provides services aimed at addressing these issues:

- fully dynamic: it defines a set of plan modification operations that are to be performed by the pDB.
- simultaneous execution and modification of the plan. It provides a generic tool to handle conflicts between plan modifications and execution.
- reduce redundancy: plan operators promoting the reuse of tasks already present in the plan

We first provide an overview of our system, and present its behaviour using an example. Then, we describe how plans are modeled in the pDB and how they are managed during execution. Finally, we describe the current implementation of this system and outline future work.

2. OVERVIEW

Fig. 1 presents the context in which the pDB is inserted. Plan generation tools (planners) are responsible for producing coherent plans, and the functional layer is a service layer which provides algorithms and interface between the software and the real world. Between these two, we introduce three components: the pDB 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. This plan is continuously interpreted by the *executive* to produce actual actions, using an event-based model. The *decision control* component is the management component which calls the planners to produce new plans for new missions, or for contingency planning for instance. It also takes the decisions needed by the executive: since our framework allows simultaneous planning and execution, we may need sometime to choose between the central plan being executed and the partial plans that are being built by the plan generation tools. This role of the decision control component will be explained in more details with the central tool for simultaneous planning and execution: *transactions*.

Let's assume that an operator needs a transport to a given point C. A robot is given this particular mission, which is an initial plan made of only one task: `Provide(what: Transport, at: C)`. This plan cannot be executed yet since there is no information on *how* this would be done. We therefore need to develop this task.

From the database point of view, planner results are a set of plan modifications which, when applied to the current plan, will produce a new global and coherent plan. The database does not check the coherency of generated plans, it provides tools to adapt them and to safely manipulate the global plan while concurrently executing it.

Since planning is in general far from being fast, the system must be able to ensure that a set of modifications is valid despite the changes brought by execution. To that end, the database provides *transactions*: this is a sandbox in which planners can freely generate the needed set of modifications *without changing the global plan*, this set being applied atomically when planning is finished. These modifications are gathered in one well-defined object such that the pDB can check their validity with respect to the global plan being executed.

In our example, a planner is selected for `Provide` by the decision control. This planner begins a transaction, and produces the set of plan modifications leading to a simple `Provide` → `MoveTo(from: A, to: C)` 2-tasks plan and applies these modifications by *committing* the transaction. Before it can be executed, however, a specific `MoveTo` modality has to be chosen. For example, our all-terrain rover has two motion modalities (Peynot and Lacroix, 2005) (Fig. 4). The NDD modality performs reactive obstacle avoidance by using a 2D laser range finder and can be used only on flat terrain. For more difficult terrain, the P3D modality uses stereovision to produce a 3D terrain model, which is then used to compute local robot path. The first one allows faster movement, while the second one is more robust. We therefore want to use NDD when the terrain allows it, and P3D otherwise.

To be on the safe side, decision control begins by selecting the P3D modality (safe, but slow). We replace the `MoveTo` task with `P3D::MoveTo(from: A, to: C)`. Note that the plan database knows that it can switch between `NDD::MoveTo` and `P3D::MoveTo` in this context, since from the point of view of `Provide`, they are both equivalent to `MoveTo`. The *replace* operator handles exchanging such alternatives.

However, changing the kind of `MoveTo` task on the fly is not that simple: we cannot have both modalities active at the same time, as it would mean that there is a way to send commands to engines from two different sources, which is forbidden in our functional layer. Therefore, when we switch motion modality there is a period of time where none is active, which breaks the plan. However, if the duration of the switch is short compared to the system dynamics, this break can

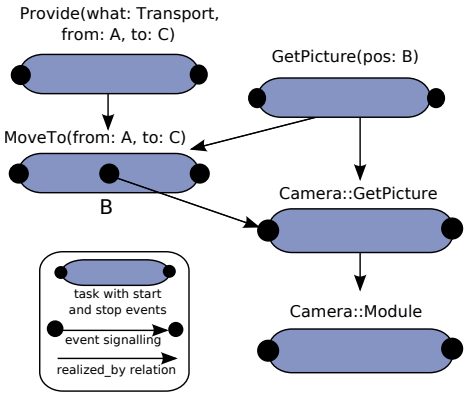


Fig. 2. Example: a robot has two goals: going to C and taking a picture at an intermediate point B. The corresponding plan is a directed graph of tasks and events.

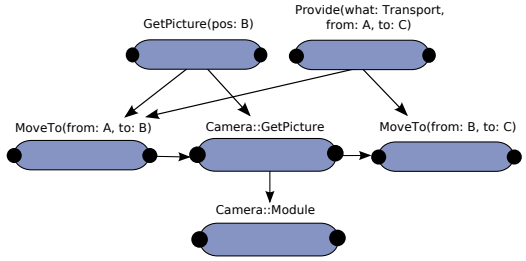


Fig. 3. The insertion of `GetPicture` into the initial plan for `Provide` requires that we split the original `MoveTo(from: A, to: C)` into two sequential motions

be allowed. The pDB defines *plan repairs* to do this in a controlled way.

Suppose now that during the execution of `Provide`, the operator decides that a picture is needed at an intermediate point B, which is not far from the planned path. The `GetPicture(at: B)` mission is inserted into the rover plan using a transaction. To allow synchronization between the movement and `GetPicture`, a new event B is dynamically added to `MoveTo` (Fig. 2). Adding this event to the task instance allows to express that we need synchronization on a particular position of the robot and that the robot should pass close to B (events are presented in more details in 3.1). If that is no longer possible, the B event enters a particular state which allows us to determine that the `GetPicture` cannot be executed anymore.

If the motion passes B before the transaction is committed, the transaction cannot be committed anymore since it depends on parts of the plan that lie in the past. Thus, the transaction becomes *invalid*. In our case, this can be solved by abandoning `GetPicture` and discarding the transaction.

But if taking the picture is an important mission, the system should try harder. Using the *split* operator, we can pause motion at B before planning `GetPicture` and later resume `MoveTo`. The resulting plan is in Fig. 3. Arbitration between abandoning `GetPicture` or pausing `MoveTo` is done in the decision control component.

Having introduced the main plan database concepts, we can define its execution cycle as follows:

(1) event propagation
(2) apply the valid transactions that have been committed to the global plan
(3) remove the set of tasks that are not required by any robot mission (unneeded tasks)

3. PLAN OBJECTS

Plans are graphs of two objects: events and tasks (Fig. 2). Events describe singular happenings during task execution and the event graph gives the information needed to perform some action when an event is observed. Similarly to TDL and other systems, tasks describe processes and the task relations provide the information needed to manage the whole set of tasks.

3.1 Event graph

Events are achievements, for instance "changed speed" or "is at position P". During execution, an event is *achieved* or *emitted* if it has already occured.

Moreover, an event is *controllable* if there is a way to make sure that it will be achieved. For instance, a `robot_stopped` event is controllable, since the system knows how to stop the robot. On the contrary, a `touched_obstacle` event is not. Controllability is implemented by attaching a procedure, called the *event command*, to an event. The event model demands that if an event command is called, then this event will be achieved sometime in the future. An event is *pending* if its command has been called, but the event is not achieved yet.

Events are linked with directed relations, which define what action to take when a particular event is achieved. Two event relations are defined: if an event e_1 *signals* a controllable event e_2, then the command of e_2 is called when e_1 is achieved. If an event e_1 is forwarded to e_2, then e_2 will be achieved as soon as e_1 is achieved.

3.2 Task models

While controllable events represent a deterministic link between an achievement and its command, tasks represent processes where simply calling the command (starting the task), does not allow to predict that the task will successfully fulfill its purpose. In the plan database, *task models* are defined as a set of events, which are the milestones of the task execution. Plans are then made of *task instances*, which are defined by a task model

and a set of parameter values. During execution, achievement of the task events is determined either internally by the task itself, or externally by forwarding external events.

All task models define the following events:

- `started`: the task is started.
- `failed`: the task has terminated, but did not realize its purpose. `failed` is controllable if the task can be interrupted.
- `success`: the task has realized its purpose. A task is *achieved* when its `success` event is.
- `stopped`: the task has terminated.

Obviously, no task event can be achieved before `started` or after `stopped`, and all of above events can only be achieved once for a given task instance. Note that all events except `started` describe a termination of the task. We therefore would want that, for instance, if `failed` is achieved then `stopped` is achieved too. Forwarding `failed` and `success` to `stopped` does that.

The definition of multiple events in the same task model allows to define multiple execution paths. For instance, one can define in a plan a signal from the `success` event of a task, and a signal from the `failed` event of the same task. Since these two events are obviously exclusive, the plan defines two execution paths.

Task models are managed in hierarchies, where a parent model defines a more generic task than a child model. Note that this hierarchy differs from the refinement hierarchy presented later.

- as tasks are defined by a set of parameters, a child model defines at least the same parameters as its parent. It can define more, but not less.
- a child model defines a superset of the events defined in its parent model

It follows that a task t_1 can be substituted by another task t_2 if its model is a child of the model of t_1 and if its parameter set is included in t_1's parameter set.

For instance, `NDD::MoveTo` and `P3D::MoveTo` are submodels of `MoveTo`, which takes two arguments: the origin and the destination. Submodels of `MoveTo` can take more arguments, such as specific parameters for the algorithms of each modality. Moreover, the system can dynamically add a new event B on `MoveTo` instances, which is achieved when the robot crosses a geometric point B. If new events have been added, then specialized task instances can be used for substitution only if the same events can be added to them.

3.3 Task instances graph

To finally form a plan, we need to describe task dependencies. Our system defines two kind of relations between tasks, which form directed acyclic graphs: *realized_by* and *planned_by*.

3.3.1. Task refinement hierarchy The *realized_by* relation defines task refinement, where a task T is *realized_by* a task t if the achievement of T requires the achievement of t, or of some of t events. In that case, t is a child of T and T is a parent task of t. The *child tree* is the tree formed by the *realized_by* relation and rooted by T. Unlike TDL or TCA, however, the tasks do not form a task tree but a *task graph*: one task can have many children and many parents, which allows to easily reuse tasks already in the plan.

The *realized_by* relation has the following parameters, which are needed for plan management:

- a $(model, arguments)$ pair which defines what kind of task T needs, following the substitution principle explained earlier.
- a set $E_{success}$ of events from the child. The child fullfilled its purpose when an event of this set is achieved.
- a set $E_{failure}$ of events from the child. Having any event of $E_{failure}$ achieved means that the child will not realize what the parent was expecting.

Normally, $E_{success} = \{\texttt{success}\}$ and $E_{failure} = \{\texttt{failed}\}$. In Fig. 2, `GetPicture` needs `MoveTo` only until B is achieved, thus $E_{success}$ is $\{\texttt{B}\}$.

3.3.2. The planned_by relation A task t is *planned_by* a task T if T represents the planning process in charge of the achievement of t. In general, it means that all or part of the subtree of t has been produced, and can be changed, by the process T. It can be used when one inserts a new mission to be realized in the future, but the system does not have enough information to plan it yet. It is then possible to simply synchronize the beginning of the planning process with the end of the tasks which should bring us that information.

4. PLAN MANAGEMENT

The pDB defines the following services for plan management:

- task transformation operators: *split*, *replace* and *merge* (*merge* is not presented in this paper as it is still a work in progress).
- handling of unexpected errors
- plan cleanup mechanism to remove unneeded tasks
- a tool to concurrently modify and execute the plan: transactions

4.1 Task transformation

The $replace(t, T)$ operator transparently replaces the task t in the plan with a new task T. This is only possible if T is a valid substitution of t for all the hierarchy relations t is part of. To replace running tasks, we need to check that the two task models are equivalent (which is done by the pDB), but also that they are in the same *execution state*. This is done internally by the two tasks: T should provide a method which ensures that its state is equivalent to the one of t, given the task models needed by t's parents. In simple cases like the `MoveTo` replacement, it ensures simply that a running task is replaced by a running task.

The $(t_1, t_2) = split(t, split_at)$ operator (Fig. 3) is used to plan the pause of a given task, and then resume it. *split* has the property that executing t_1 and t_2 in sequence gives the same result that the whole task t, and that in between t_1 and t_2, we can assume that t is not running. In practice, this kind of plan modification may be problematic during task removal: if the `Camera::GetPicture` is removed during plan execution, we shall merge the two tasks back into the original task. This problem will be handled by merging mechanisms that are still under development.

4.2 Error handling

We want to be able to express plans where failure is explicitly taken into account. Therefore, like ESL (Gat, 1997), and unlike TDL (Simmons and Apfelbaum, 1998), failures are not always exceptional conditions. Separation between different failure modes is then done by defining new events which are forwarded to `failed`. Therefore, the model does not consider that all non-nominal events are errors. While TDL would for instance handle a failed movement as an exception (Simmons and Coste-Manire, 2000), this is an error so common in unknown environments than its correction should have already been planned. However, if it is not the case, the executive looks for an error handler in the task refinement hierarchy and the planning tasks. If no handler is found, the failed task is removed.

Another central tool for for error management allows to *asynchronously* repair the plan: one can register a task as repairing some parts of the plan (for instance handling the `failed` event of `MoveTo`). The plan can then remain broken at this failure point. A timeout ensures that it remains so only for a period of time short with respect to the system dynamics: the executive considers that the plan is being repaired as long as the task is running and will not raise the corresponding exceptions. If the plan is not repaired when the task is finished, or when the timeout is achieved, normal exception handling is resumed.

4.3 Plan cleanup

During the lifetime of the system, tasks will be inserted and removed dynamically. We need to be able to determine what tasks can be removed, given that we know what are the high-level goals of the robot: if we remove a high level task like `GetPicture` in Fig. 2, we need to remove *some* of its children (in this case, `Camera::TakePicture`). There are therefore two kinds of tasks in a plan. The *missions* are the high-level goals of the robot, while the remaining of the plan exists to achieve these missions. It follows that the plan can be split into a set of tasks needed by the set of missions, and a set of tasks that are not useful in the context of the current missions. The first set is part of the child tree of a mission, while the second is not. Note that a mission is in general not explicitly removed by an external tool: it is marked as not being a mission anymore, and the system will clean it up itself.

Planning tasks are handled differently. Since a planning task objective is to produce a plan for a set of tasks in the plan, they can be cleaned up only if all the tasks they are planning are marked as being removed as well.

While tasks that are not running can be simply removed, running tasks should be stopped before they are removed. Since their `stopped` event command can handle both the end of the task itself, but also of its child tasks, we can't remove a task which is in the realization tree of a running task. Therefore, the cleanup algorithm is as follows:

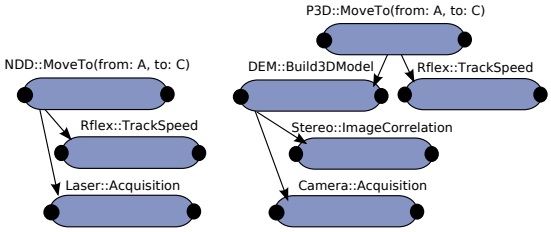
(1) Remove all tasks in the plan which are not needed anymore, which have no parent task and which are not running.
(2) Repeat until there are no more such tasks.
(3) Compute the set of tasks which are not needed anymore and which have no parent.
(4) For all tasks in this set that have a controllable `stopped` event, call that event.

While cleanup is running, no event is propagated and no plan modification is done.

4.4 Concurrent plan modification and execution

One of the main contributions of this plan management system is to provide the tools needed for safe modifications of plans during execution.

While one can modify the global plan directly, it is dangerous as the executive would begin the execution of some parts that are not finished yet. Currently, most executives solve this problem by forbidding the modification of the short-term plan. We find this solution quite limited since a low-level executive shall often do reactive modifications of the plan. Our pDB offers *transactions*, which gives a context to build a set of plan modifications outside of the global plan, subsequently



(a) `NDD::MoveTo` modality (b) `P3D::MoveTo` modality

Fig. 4. Task refinement of motion modalities. Because the two modalities need the `Rflex` to control the robot motors, they can't be running at the same time.

committed either all at once or not at all. The transaction is checked before commit for soundness: the commit is therefore only an application of plan changes, and thus is not a CPU intensive task. Transaction commits can thus have their own slot in the execution cycle, which means that no event propagations is done during the commit.

As we saw in section 2, concurrent execution and planning can lead to *invalid* transactions: the transaction depends on the `MoveTo` task, which is running while in the main plan while the transaction is being built. If during planning the B event, which is depended-upon by the tranasction, is emitted the transaction becomes *invalid* and cannot be committed. An invalid transaction can be discarded or repaired by changing the transaction, the global plan, or both. Choosing the way to handle transaction invalidation is done by the decision control component, whose role here is two choose between a modification of the transaction or a modification of the plan, by calling plan generation components if needed.

In this example, the B event is emitted but its command (if it would have one, which is not the case) has not been called by the executive. We may have situations where the transaction is invalidated because of plan signalling: for instance, if a signal was to terminate the `MoveTo` task by calling its `failed` event, decision control would have to choose between doing this signal and invalidating the transaction, or forbidding the signal. If the signal is chosen, the transaction is invalidated and the resolution procedure is the same as before. However, if the transaction is chosen, an exception is thrown from the signal source, whose effect would be to either change the plan so that the `MoveTo` task shall not be stopped, or remove the part of the plan which was relying on this signal.

5. CONCLUSION

The pDB is currently implemented in the Ruby language, on top of the GenoM [1] (Fleury *et al.*, 1997) functional layer, and will be used on both a rover and a blimp. Plans are currently produced using a simple script-based plan generator similar to the plan generation capabilities of TDL. Moreover, a multi-robot interaction scheme is in development to meet the needs of our experimentation. This scheme bases interaction on the capabilities of an agent to provide information which can facilitate the execution of another agent tasks: in our experimentation, the blimp will provide traversability information to the rover.

One of our research objectives is to allow the simultaneous use of multiple planners in the same system, in order to use the more efficient planner according to current needs. This cannot be achieved without plan merging mechanisms. The development of a *merge* operator is an ongoing work that is based on the presented substitution principles, event and task structure and an additional conflict predicate for tasks that can not be executed in parallel.

The contributions of this paper are the plan model and the set of plan management mechanisms built around it. The pDB allows insertion and removal of tasks, using the task hierarchy to detect the consequences of the removal operation. It also provides a mechanism to safely modify a plan online, *transactions*, which guarantee that there are no partial modification of the global plan.

[1] available at `http://softs.laas.fr/openrobots`

REFERENCES

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

Finzi, A., F. Ingrand and N. Muscettola (2004). Robot action planning and execution control. In: *Proceedings of IWPSS*.

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

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

Ingrand, F., R. Chatila, R. Alami and F. Robert (1996). PRS: A high level supervision and control language for autonomous mobile robots. In: *Proceedings of IEEE ICRA*.

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

Simmons, R. and D. Apfelbaum (1998). A task description language for robot control. In: *Proceedings of the Conference on Intelligent Robots and Systems*.

Simmons, R. and E. Coste-Manire (2000). Architecture, the backbone of robotics systems. In: *Proceedings of IEEE ICRA*.