A generic framework for anytime execution-driven planning in robotics

Florent Teichteil-Königsbuch
ONERA / DCSD
31055 Toulouse, France
Florent.Teichteil@onera.fr

Charles Lesire
ONERA / DCSD
31055 Toulouse, France
Charles.Lesire@onera.fr

Guillaume Infantes
ONERA / DCSD
31055 Toulouse, France
Guillaume.Infantes@onera.fr

Abstract— **Robotic missions require to implement various functionalities in order to link reactive functions at actuators and sensors level to deliberative functions like vision, supervision and planning at decisional level. All these functionalities must be versatile and generic enough to interact differently according to the missions while minimizing recoding effort. Moreover, deliberative functions like automated planning consume lots of memory and CPU time and usually complete in time incompatible with robotic missions' durations. Thus, we present a new generic and anytime planning concept for modular robotic architectures, which manages multiple planning requests at a time, solved in background, while allowing for reactive execution of planned actions at the same time. Different planners based on various formalisms and data structures can be plugged to the planning component without changing its behavior nor its code, facilitating reusability and validation of the component. We highlight the versatility of our concept on different use cases; then we demonstrate the efficiency of our approach in terms of mission duration and success, compared with traditional plan-then-execute approaches ; we finally present a search and rescue mission by an autonomous rotorcraft solved with our paradigm, that cannot be tackled by traditional approaches.**

I. Motivations

Autonomous robotic applications require to implement various algorithms from control laws to vision algorithms, including high-level sequential decision-making known as *automated planning*. While designing such algorithms to solve realistic robotic missions is a very challenging area of artificial intelligence, embedding and connecting them on a software architecture for real-time execution is an other ambitious topic. Indeed, deliberative reasonings like vision and planning consume a lot of CPU time, which is generally far higher than durations of missions and robots' actions, so that the underlying algorithms cannot be simply embedded in the system as such.

Yet, software architectures should ideally provide a mechanism to integrate these deliberative algorithms without requiring to modify them, since they are often designed for more general purposes than execution on-board a robot. In case of planning algorithms, we claim that a component of such an architecture should be conceived as a generic kernel managing interactions with other functionalities to which many planning algorithms can be plugged without changing its code nor its behavior. This article precisely proposes a formal definition as well as a real implementation of such a planning component for modular robotic architectures, which satisfies three main requirements: *reactivity*, *genericity*, and *validation*.

A. Illustrative autonomous mission

In order to illustrate the needs, we consider a search and rescue mission: a human being is at risk in a remote place, after a plane crash in an enemy zone. We want to send a helicopter UAV, able to scan autonomously the area, detect the human being, land autonomously at the proper place and escape the danger zone with the human being safe and secure inside. Such a mission needs online planning capabilities: the helicopter has to scan then select, given the first mapping of the zone, the area where to go, search for the human being, and possibly land. It has also to plan its trajectory between waypoints while avoiding obstacles or forbidden areas. More precisely, this mission is monitored by a *supervisor*, that controls all the components of the architecture: the *navigation* component controls the helicopter movements and provides state information (pose, velocity); the *camera* component manages image acquisition; the *mapping* component implements mapping algorithms that build an obstacle map given acquired images and their corresponding state vectors; the *zoning* component extracts zones from the map where the helicopter could possibly land.

This mission raises several points: first, the planning problem cannot be solved off-line before the mission starts, because the UAV needs to scan the area before selecting zones to explore precisely. This prevent a priori building of a policy. Furthermore, the number of zones may be very large, and precise management of resources like fuel and memory (to store and process images) is mandatory, leading to a potentially very large state space, while rescuing time has to be as small as possible[1]. Finally, the uncertainty level is very large for this data, making exhaustive a priori planning intractable. So, as the planning times may not be guaranteed, and cannot be considered instantaneous, correct interleaving of planning and execution is a critical part of achieving such a complex task. One approach is to have the supervisor launching planning requests to the deliberative planning component and uses the results when/if needed.

B. Generic use cases

In a more general perspective, we can see at least three different ways for using the planning component considered as a deliberative part in relation to a reactive supervisor.

[1]The state space is exponential in the number of zones with continuous variables, so that optimal optimization under uncertainty with state-of-the-art planning algorithms would last many centuries with a 1Ghz processor.

a) Pro-active planning: In real-world scenarii, the planning delay is to be taken into account: the plan is executed starting from a future state which is different from the present one, for instance if the fuel level, or remaining time to save the human being are part of the system state. This problem is often eluded by computing a policy (a conditional plan, where the action to do is a function of the state) instead of a plan. But computing a complete policy can be very long if tractable at all, and many algorithms use a start state in order not to explore the whole state space. This is especially true in classical planning [1], but also in recent MDP planning [2], [3], [4]. So a realistic setup may use a state predictor (or prognostic) giving a set of possible states for time $t + \delta_{planning}$, where t is the present time, and $\delta_{planning}$ is the delay allocated to planning. So the planning component has to handle several requests for several starting states at once. Depending on the state predictor and the degree of uncertainty of the evolution of the system, the start states may be close to each other, or very different.

b) Cooperation: Another use case can appear if the previous scenario is augmented with other agents needing to know how the autonomous UAV may react in some circumstances. For instance, another UAV may be trying to protect the first one, and needs to know where it may go if a zone is found dangerous. This case may also be useful for verification and validation needs.

c) Long term anticipation: A more subtle case appears if the model and/or the algorithm is not fully trusted. For instance, if there exists a fault that is very unlikely to occur given the current model, then the planning algorithm may not explore it (for instance if it uses state sampling with few samples), thus may not be able to give a good policy for such faulty states. In that case, the supervisor may anticipate such cases in order to store a quick reaction by asking the planner what to do if in such unlikely-to-occur states.

In this paper, we propose a generic and reactive scheme for using the planning component in order to be able to deal with complex problems for which execution and planning have to be properly interleaved. We call this scheme "execution-driven" because it is totally different from the classical "plan, execute then replan" scheme. We claim that for most real-world problems, the goals do not change much (especially if we consider optimization of a policy where multiple conditional goals are possible), but the states from which we need to execute (or know the action that would be executed) do. Furthermore, we propose to separate the planning component from the planner itself, so that the planning component is generic in the sense that it can be used with different, specialized planning algorithms.

II. AUTOMATED PLANNING FOR ROBOTICS

Designing efficient automated planning [5] algorithms requires to define the problem and its hypotheses precisely for best performances. However, the planning component proposed in this paper totally separates planning requests from plan computations, so that a general definition is sufficient for our needs: we consider a planning problem as an automatic generation of action sequences or strategies that achieve a given numeric or symbolic objective, knowing the current state of the world and actions' effects. Whatever the computations involved by planning algorithms, we define a unified framework to execute optimized actions reactively to environment changes. The following definitions will be used as data flows by the planning component but defined by each possible planning algorithm plugged to the component.

A. Basic concepts

Our generic planning component described in the next section relies on a few generic concepts common to most planning algorithms:
- $\mathcal{P}$: a planning problem;
- $\mathcal{S}$: a set of states; $s \in \mathcal{S}$: a stat ;
- $\mathcal{A}$: a set of actions; $a \in \mathcal{A}$: an action;
- $App : \mathcal{S} \rightarrow 2^{\mathcal{A}}$: function indicating all actions applicable in a given state;
- $T : \mathcal{S} \times \mathcal{A} \rightarrow 2^{\mathcal{S}}$: transition function indicating all possible next states when applying a given action in a given state;
- π_{s_0}: a solution of the planning problem $\mathcal{P}$ for a given initial state s_0 (more on this below);
- $\pi_{default}$: a default solution of the planning problem $\mathcal{P}$ defined over all possible states reachable during the mission;
- $alg_{\mathcal{P}}$: an algorithm that solves the planning problem $\mathcal{P}$;
- $param_{alg}$: tuning parameters for algorithm alg.

If states are only partially observable [6], actions are optimized and executed for given *belief states*, which can be probability distributions over the actual state. To cover this case, we can assume that $\mathcal{S}$ is the set of belief states. In this case, actual hidden states can not be accessed through the planning component, and, in such a setup, estimation of the current belief state from observations (filtering) is done outside the planning component. This is realistic as it can be seen as a special case of the first use case, because state estimation is needed for robotic application, and can be seen as a sub-problem of state prediction.

Another important concept is the solution of a planning problem. If actions' effects are deterministic [5], [1], a sequence of actions named *plan* is sufficient to reach the problem's goal. However, in other cases such as probabilistic [7], [2] or non-deterministic [8] effects, the execution of a plan may lead to states where no actions are defined. If an approach based on replanning is too slow for the target application, the planner has to produce a *policy* function $\pi_{s_0} : (\mathcal{S}_{reach(s_0)} \subset \mathcal{S}) \rightarrow \mathcal{A}$ indicating the action to apply in states reachable from an initial state s_0.

Conveniently, any plan can be represented by a policy, so that the latter concept embraces most solutions of any planning problem. We argue that the policy concept also is interesting for planning with deterministic effects, since it allows the planning component's caller to know if the planner's model is deviating from the model of the world, by asking if current state is in the policy: if not, it means that the model of actions' effects is wrong.

The default policy, owned by the planning problem as default solution of it, is mandatory for reactivity and validation of the planning component. It guarantees that a default action is always available whatever the state of the world, even if the planner could not find a solution for this state. Moreover, on condition that it was validated before being embedded in the planning problem, the default policy ensures a safe action to apply that does not end up in some dangerous feared state in the future. In most cases, this default policy can be defined over sets of states instead of individual states, so that it is compact enough for embedding.

The previous concepts are implemented outside the planning component, in each planner plugged to the component. This way, we can design a generic planning component whose requests' management and implementation are independent from planners' data structures and common to all planners. All these concepts are totally abstract from the planning component's point of view: for instance, if actions' effects are probabilistic (the MDP case [7]), the set of possible next states (and hence the concept $\mathcal{S}$) will be actually defined by the planner as a probability distribution over states. In the same line of thinking, the algorithm alg and its parameters $param$ only affects the planner plugged to the component to solve the planning problem, so that they are defined in the planner but not in the component.

B. Requests definition

As the planning component provides services about a given planning problem to other components, its core is all about managing requests concerning the problem, independently from available algorithms and data structures needed to solve it. We identified five requests whose final processing is delegated to the planner plugged to the component:
- *loading a planning problem $\mathcal{P}$*: constructs all data structures needed to solve $\mathcal{P}$;
- *getting the actions $A_s \subset \mathcal{A}$ applicable in state s*: constructs the set of actions applicable in s (this operation can be processed only by the planning component if it requires data structures constructed inside it);
- *getting the effects $E_{s,a} \subset \mathcal{S}$ of action a applicable in state s*: constructs the set of states that may be immediately reachable by applying the given action in the given state;
- *computing the policy π_s solution of the loaded planning problem from the initial state s during t seconds with algorithm alg and parameters $param_{alg}$*: constructs a partial solution in bounded time from a given state and inserts it into the policy function computed so far;
- *getting the action a_s planned in state s*: reads in the policy function the action planned in s and returns it.

We claim that these five request classes are sufficient in order for the planning component to satisfy at least the requirements of the use cases presented in section I. Furthermore, these requests are versatile and general enough to deal with other situations, provided they can be non-blocking for the calling component and achieved in parallel. This point is all about the behavior of the planning component, detailed in the next section.

III. A GENERIC AND ANYTIME PLANNING FRAMEWORK

The planning component detailed in this section is designed to fulfill the high-level requirements of section I:
- *reactivity* to environment changes through other components' requests, achieved by: (1) parallel execution of plan construction and requests' management; (2) default policy returning a solution in short bounded time if the planner could not find a solution for a given state; (3) state machine formalizing the interactions with the planning component.
- *genericity* with regards to planning algorithms and functional architectures, achieved by: (1) plugin system: the planning component is a class template whose variable type is a planner; (2) separation between requests' management in the planning component class and plan construction in the planner class; (3) definition of a fixed interface with other components through five request member functions.
- *validation* of the component's code and behavior, achieved by: (1) validation of the planning component, assuming safety conditions on the planning algorithm side; (2) validation of the planning component's default policy for a given mission; (3) validation of each planner plugged to the component (future work).

The next paragraphs look deeper inside the planning component to understand how we designed and implemented the above items.

A. Component's states

The state machine in Fig. 1 defines the different possible states of the planning component, and transitions between states depending on requests from other components of the architecture. All requests of section II-B are managed. The initial state is `Waiting Problem`, so that the component must be initialized by loading a planning problem.

After the problem has been loaded, the planning component can receive *planning requests* from the `add_plan_request` method. They can be canceled with the `remove_plan_request` method, in order to save time if for instance some initial state has been predicted earlier, but the robot is now deviating from it. Whether the state machine is in the state `Planning` or `Problem Solved`, it is always possible to get the applicable actions (`get_actions` request), the planned action (`get_action` request), or the effects of an action (`get_effects` request), for a given state. Finally, the planning component can be stopped at any time with the `stop` request, even if it is loading or solving the problem.

From the point of view of the calling component, no request is blocking: all requests immediately return a boolean, indicating if the request could be performed in the current state of the planning component. If computations are required (for loading a problem or solving it from some initial states), the state machine enters a state whose name indicates that the planning component is processing them (`Loading Problem` or `Planning`). As soon as the computation is finished, the state machine automatically enters a state pointing out that the computation is done (`Problem Loaded` or `Problem Solved`).

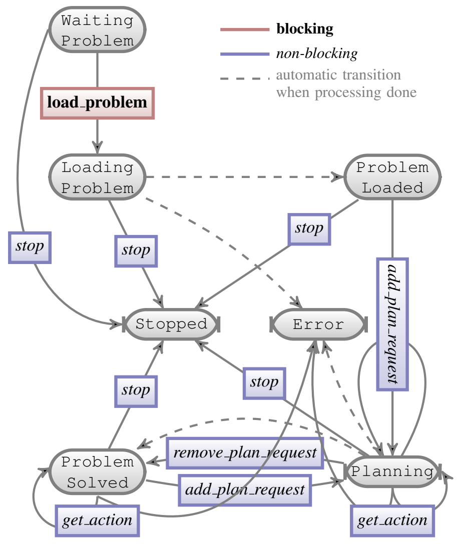

Fig. 1. State machine of the planning component. Not all requests are represented for readability reasons.

The planning component is always responsive to requests, except when loading a problem (it is prohibited to load two problems at the same time or to manage requests about a problem whose loading has not yet terminated). In other cases, the component is able to process requests and solve the planning problem in parallel. This requires to protect the policy by mutual exclusion as explained in the following.

B. Requests management as transitions

Figure 2 illustrates the management of requests arriving in the planning component. The latter contains an instance of a planner in charge of all computations related to the planning problem. The planner owns data structures required to construct the plan, i.e. the instantiated planning problem and the policy function. The `load_problem` request triggers the generation in the planner of data structures called *planning problem*, needed to construct plans. Once generated, the `get_actions` and `get_effects` requests can ask the planning problem for applicable actions or effects of an action in a given state.

Requests to improve or expand the policy by considering new initial (computation) states are queued in a list of *planning requests* each time a `add_plan_request` arrives and can be removed from the list with `remove_plan_request`. As long as the queue is not empty, the first element is popped out, and it triggers a new plan construction process in the thread of the planner when the previous plan construction has finished. This process is divided in small successive computation chunks corresponding to the `solve_progress` function, which computes and copies to the planning component the planned action for the state of the planning request being processed. The `get_action` request asks for the planned action to apply in a given state by reading in the copied policy. This request is filtered by a default policy in order to always provide an action to apply, even if the copied policy does not contain such an action (because of errors or not enough time to compute it).

Remember that problem solving and policy execution (requests management) are processed in different threads, what involves to take some thread-safety precautions. In other terms, no readings or writings in a same data structure should happen at the same time. First, the planning request queue is protected by mutual exclusion to prevent from adding or removing planning requests in parallel. Second, the `get_actions` and `get_effects` requests safely read in the planner's planning problem because, as formalized in the state machine of Fig. 1, these requests are only allowed after the planning problem has been loaded (only time where the problem is written). Last but not least, the `get_action` request reads in a local copy of the planner's policy, so that there is no runtime conflict between the thread writing in the planner's policy and the one reading in the copied policy, provided the latter is protected by mutual exclusion. So, the last point means that the code of the planner does not need to be thread-safe, what is crucial to integrate any available planner without changing its code nor the one of the planning component. It contributes to make our planning component generic while safe, along with a simple but expressive planner interface as presented in the next section.

C. Planners' interface

The planning component is an object-oriented template class that contains a variable planner. The interface of the latter must define some types and methods, like *load_problem, solve, converged, get_action, get_effects*, otherwise the compiler complains that the planner violates the template contract. The embedded types of the planner correspond to the concepts listed in section II-A.

The member functions are called by the planning component to process the requests it receives. Some of them are small computation chunks of time consuming processes (problem loading and problem solving), executed in the same thread, but different from the thread of the requests:
- `<process>_begin`: initializes computation;
- `<process>_progress`: executes one computation step;
- `<process>_end`: terminates and cleans computation.

This way, if necessary, both processes can be interrupted before they have completed. It is particularly useful when the planning component receives a request to remove the planning request being processed, so that the latter can be aborted before completion.

IV. Experimental results

A. Implementation on the Orocos platform

Orocos [9] is a C++ library for robotics focusing on real-time support and component-based programming. It allows to create distributable components and to guarantee real-time and thread-safe communications. Each component has a standardized interface, which describes its data ports and the services it provides. A component can react to events, process requests, and execute scripts in real-time.

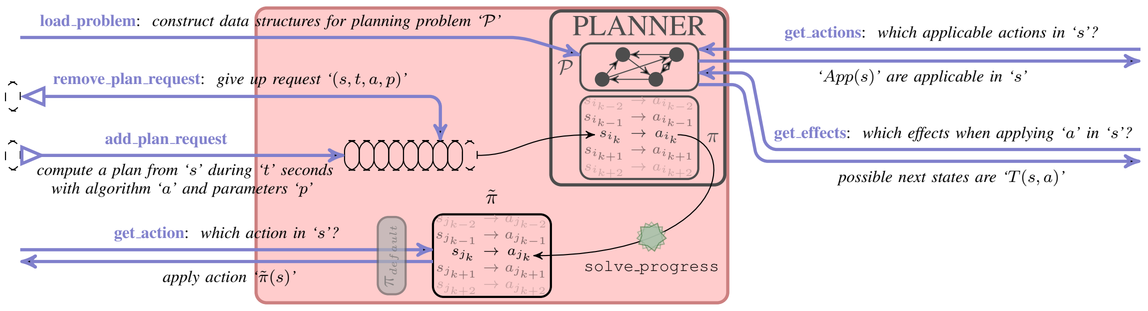
load_problem: construct data structures for planning problem 'P'
PLANNER
get_actions: which applicable actions in 's'?
'App(s)' are applicable in 's'
remove_plan_request: give up request '(s, t, a, p)'
add_plan_request
compute a plan from 's' during 't' seconds with algorithm 'a' and parameters 'p'
get_effects: which effects when applying 'a' in 's'?
possible next states are 'T(s, a)'
get_action: which action in 's'?
apply action 'π̃(s)'
π_default
solve_progress

Fig. 2. Inside the planning component: management of incoming requests and outgoing replies.

For the next test beds, we implemented two Orocos components: our planning component that optimizes strategies[2], and a simple supervisor component that queries the planning component for the optimal action in the current state and executes it (in parallel of planning). Once the problem has been loaded on the planning component, the supervisor successively: (1) asks the planning component for the action to perform in the current state; (2) starts executing this action (each action is performed by a specific state machine); (3) predicts the future possible mission states (using the `get_effects` request of the planning component) and requests for plans corresponding to these states; (4) when the action execution is done, gets the current state and goes back to point (1) until the mission ends. This way, actions are executed while the strategy is optimized for the next possibles states: we name this specific use case of our planning component *plan-while-execute*.

B. Stochastic planning missions

In order to assess the efficiency of our *plan-while-execute* paradigm (`ON` for short), we tested stochastic planning missions modelled as Markov Decision Processes (MDPs, [7], [4]) whose complexity is known to be prohibitive for large robotic problems. We tested many problem sizes (number of states in the MDPs) and mission durations so that a goal state must be reached as fast as possible before the end of the mission. For each test, we generated 100 random problems, solved them with different MDP algorithms, and simulated 100 times the execution of the policy of each problem and each algorithm. We then averaged the results over random problems, algorithms and simulated executions. We compared our approach with the traditional *plan-then-execute* paradigm (`OFF` for short), where the policy is computed until convergence before execution or possible re-planning in case of execution failure. The RTDP algorithm [2] was also tested, since its anytime design fits well with our *plan-while-execute* paradigm; we also want to compare our paradigm with out-of-the-box anytime approaches like RTDP.

Results are shown in Fig. 3, commented in the following paragraphs from left to right:

Total mission time. Whereas `OFF` is best for small missions (100 to 10000 states), `ON` achieves missions in far less time for large problems (100000 and 250000 states) and with a smaller proportion of planning time, what spares a lot of on-board power energy. `RTDP` within our planning component is better than `OFF` but worse than `ON`, what proves that our online planning request management framework (even with convergent algorithms!) allows for richer and more efficient interleaved planning than simple anytime algorithms.

Average plan length. `ON` reaches the goal state with larger steps than `OFF`, since strategy optimizations for the next possible states are stopped (before convergence) as soon as the current action finishes (via `remove_plan_request`), so that stategies are not optimal. `RTDP` is the worst because of its very slow convergence.

Success, time-outs and default actions. For all approaches, the percentage of success decreases with the mission duration, which is however chosen so that the mission is achievable with the optimal plan length: for `OFF`, planning time is too large; for `ON`, plan length is too large (since not optimal). Yet, `ON` always achieves far more missions than `OFF`, especially for very time-constrained missions. Logically, the number of time-outs is larger with `OFF`. Because of its slow convergence, `RTDP` uses more default actions than `ON`.

To summarize, for large problems and time-constrained missions, our generic planning request management framework achieves more missions in less global time than traditional approaches, but with a slightly reduced optimality.

C. Emergency landing mission

We also tested and embedded our planning component on-board a real autonomous helicopter for the emergency landing mission described in section I-A. Two planners have been implemented: a PathPlanner component, that computes paths to reach specific points or to explore a zone; and a MDP-based MissionPlanner component, that optimizes high-level actions (such as 'land', 'goto x', 'map zone z') to achieve the mission. Both planners' data structures are totally different, but they share a single code for managing all planning requests.

Traditional *plan-then-execute* approaches cannot achieve the mission, since the strategy converges in many centuries, long after the human is dead. The autonomous helicopter embedding our planning component in a realistic outdoor environment rescued a fake human in roughly 10 minutes.

V. Related work

While no generic planning library for robotics is available today, other robotic architectures rely on the interleaving of

[2]The source code of our generic Orocos-based planning component along with some already plugged planners is publicly available at `http://robotis.onera.fr/orocos/planning/`.

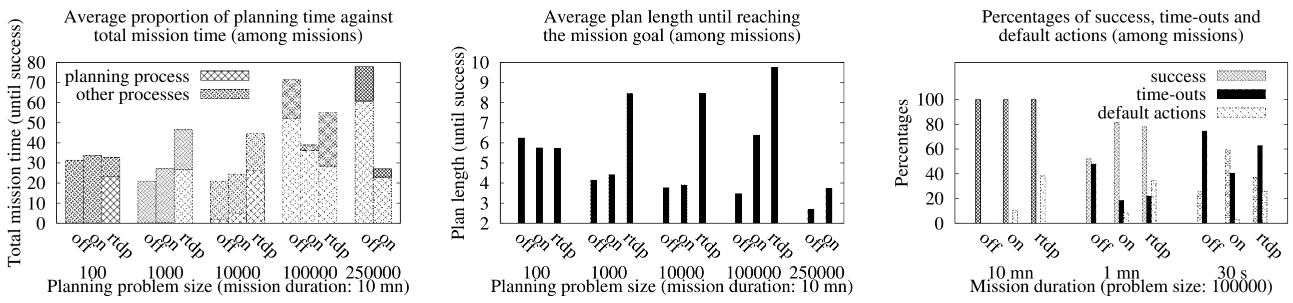

Fig. 3. Solving random stochastic planning missions: comparison between using our generic planning component, either with VI [7] and LAO* [3] algorithms ("on") or with the RTDP [2] algorithm ("rtdp"), and traditional *plan-then-execute* approaches with VI and LAO* algorithms ("off").

planning and execution. The framework proposed in [10] is based on a central Plan Manager that interacts with a planner component by sending planning requests. The produced plans are stored in a database where the Plan Manager can look for appropriate actions. The Plan Manager ensures the overall reactivity by providing an action whatever the plans in the database. While genericity seems a major concern of this approach, no generic behavior of the planner component is described and no programming framework is provided.

The architectures proposed in [11], based on continuous planning, or [12], [13], based on temporal planning, interleave decision and execution by updating the plan from the current state. However the planner can only compute a plan on a unique situation and cannot deal with several alternative requests corresponding to different possible future situations. Hence a major change in the next state may need a complete replanning, delaying action execution.

IDEA [14] and T-REX [15] are agent-based architectures where each agent has its own behavior based on timelines, and possibly includes a local planner. Each planner is then specialized, responsible of a specific problem (global mission objectives, navigation, guidance…). Each agent is then in charge of monitoring its own actions and replanning its local timelines. Whereas such a fully decentralized approach is possible in the framework we have proposed (the architecture can embed multiple planning components), a single planning component can handle multiple situations (in a unique problem) and compute the associated policy. On the opposite, the IDEA/T-REX framework could allow such a behavior by instantiating one planner agent per situation, increasing the complexity of agents' and timelines' interactions.

All these approaches consider reactivity issues as part of the executive component. However, such a component cannot be fully reactive if it has no way to request for pro-active plans, applicable in a large sample of possible situations. The framework we have presented allows to help the supervisor be more reactive by providing a reactive behavior of the planner, based on multiple requests, time-bounded computations, and the availability of a default policy.

VI. Conclusion

In this paper, we presented a formalized concept of an anytime execution-driven planning component for robotic modular architectures, focused on three requirements: reactivity, genericity and validation. Reactivity means that the component is able to process planning requests on future possible states in advance, while immediately replying to any action requests on a given state. Genericity intends that the planning component is designed for being used in various missions connected to different components, and that its behavior and code is parted from planning algorithms, which are delegated to variable unknown planners. Validation results from genericity in the sense that the planning component can be validated once and for all, assuming satisfied properties to validate for each planner plugged to the component. We demonstrated our planning component's reactivity and genericity on many challenging robotic problems. We intend to formally validate it in the future.

References

[1] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *JAIR*, vol. 14, 2001.
[2] A. Barto, S. J. Bradtke, and S. P. Singh, "Learning to act using real-time dynamic programming," *AIJ*, vol. 72, pp. 81–138, 1995.
[3] E. A. Hansen and S. Zilberstein, "LAO* : A heuristic search algorithm that finds solutions with loops," *AIJ*, vol. 129, no. 1-2, pp. 35–62, 2001.
[4] F. Teichteil, U. Kuter, and G. Infantes, "Incremental plan aggregation for generating policies in MDPs," in *Proc. AAMAS*, 2010.
[5] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann, 2004.
[6] L. Kaelbling, M. Littman, and A. Cassandra, "Planning and acting in partially observable stochastic domains," *Art. Int.*, vol. 101, 1998.
[7] M. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, Inc. NY, USA, 1994.
[8] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, "Strong planning under partial observability," *Art. Int.*, vol. 170, no. 4-5, 2006.
[9] P. Soetens and H. Bruyninckx, "Realtime hybrid task-based control for robots and machine tools," in *Proc. ICRA*, 2005.
[10] K. Myers, "CPEF: continuous planning and execution framework," *AI Magazine*, vol. 20, no. 4, pp. 63–69, 1999.
[11] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau, "Using iterative repair to improve the responsiveness of planning and scheduling," in *Proc. AIPS*, 2000.
[12] S. Lemai and F. Ingrand, "Interleaving temporal planning and execution in robotics domains," in *Proc. AAAI*, 2004.
[13] P. Doherty, J. Kvarnström, and F. Heintz, "A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems," *JAAMAS*, vol. 19, no. 3, pp. 332–377, 2009.
[14] A. Finzi, F. Ingrand, and N. Muscettola, "Model-based executive control through reactive planning for autonomous rovers," in *IROS*, 2004.
[15] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen, "A deliberative architecture for AUV control," in *Proc. ICRA*, 2008.