Towards Automatic State Recovery for Replanning

Stefan-Octavian Bezrucav[*1], **Gerard Canal**[*2], **Andrew Coles**[2], **Michael Cashmore**[3], **Burkhard Corves**[1]

[1] Institute of Mechanism Theory, Machine Dynamics and Robotics, RWTH Aachen University
[2] Department of Informatics, King's College London
[3] Department of Computer and Information Sciences, University of Strathclyde

Abstract

Integrated planning and execution on embodied agents necessarily means dealing with execution failures, and it is common to employ on-board replanning to overcome such failures. However, describing the activities of the agent as action definitions appropriate for planning often requires a discrete abstraction of continuous state. As a result, action failure can result in "improper" planning states that are inconsistent with the ground truth. Attempting to generate a plan in these states can result in planning failure, even in cases where the executive actually has the capability to reach the goal.

In this paper, we formalize the concept of *proper* states, the mappings between planning states and underlying execution structures, and lay the grounds towards developing automatic recovery behaviors when execution failures leave the execution in an artificially incorrect planning state.

Introduction

Planning is used to generate and order a set of actions required to transform a given system from an initial state to a state where some goals are achieved. The real world is complex, and in order to make the problem manageable the model used for planning is an abstraction. Often the planning problem includes a discrete representation of what is in reality a continuous space. In addition, lower-level executors must be implemented for each action to execute the plan, which might also abstract the problem into a discrete space.

As the planning and the execution levels might work with different representations of the scenario (e.g., in granularity and abstraction level of the state spaces), discrepancies might occur between them, especially when execution fails. This paper introduces a formal method for handling these discrepancies and to enable replanning in case that an action fails. In this paper, we consider only planning problems formulated in the Planning Domain Definition Language (PDDL) (Fox and Long 2003).

Figure 1 shows an example of the planning and execution processes in different state spaces, levels of granularity, and abstraction. First, task planning is performed on a highly abstract representation of the scenario. Only relevant state space details such as the two possible locations are encoded in the planning problem, while the continuous space between them is not modelled. On the planning

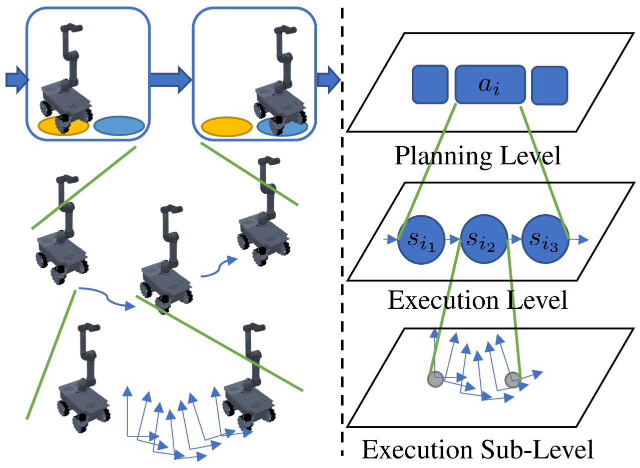

Figure 1: Planning and execution on different levels of abstraction and the mappings between them.

level, the plan has one move action. On the execution level, each of the high-level actions is described by a Finite State Machine (FSM), with its associated state-space representation. In this case, the move executor contains two navigation states. These FSMs encode more information about the action, but they are formulated only for one part of the scenario (e.g. for one action). The execution of an action can then be described on more levels.

The low-level executors will achieve parts of the planning action's effects during the execution rather than applying the effects at start or at end as it happens in temporal planning (or all of them at end of the action as in classical planning). Thus, planning and execution with different levels of abstraction becomes challenging when errors occur during the execution of an action. In such cases, the typical "plan-fail-replan" loop might break since a replan may happen when only a part of the expected effects have been applied. This may result in inconsistent or *improper* planning states that may prevent the planner from finding other viable solutions.

In this paper, we formalize the problem of inconsistent states that can appear in PDDL planning and execution in different state spaces. We define states that allow replanning to be performed when a low-level executor has failed. The

[*]These authors contributed equally.

proposed approach also defines mappings between the planning and the execution levels. These mappings allow to statically check where an interrupted execution would lead to an unrecoverable state from the planning level. This methodology aims to increase the robustness of plan execution.

Related work

Suitable approaches for generating a new plan when the actual one either has failed or the goals have changed have been analysed in many works in literature. Nebel and Koehler (1995) compared the advantages of replanning from scratch and of reusing information from the old plan to generate a new one, when this old plan does not anymore correspond to the planning scene. The theoretical analysis of the authors showed that the second solution, although more intuitive, is not necessarily more computation effective. The work of Fox et al. (2006) argues that plan repair strategies can be developed that generate plans more efficiently than a replan strategy. Further, such repair approaches help to maintain the plan stability, that is, how close the newly generated plan is to the one that it must replace. Further results show that, in practice, plan adaptation can be more effective than re-planning (Gerevini and Serina 2000; van der Krogt and de Weerdt 2005; Cushing, Benton, and Kambhampati 2008). However, the proposed approaches deliver good results especially when the new initial and the goal states do not drastically change. Further, most approaches modify the planning strategies within existing planners.

An alternative to modifying the planning strategies within the planners is to incorporate the planner in a framework. This framework integrates the planning and execution in a control loop and coordinates the replanning procedures. A replan means a new call to the same, unmodified planner for a different planning situation. To work properly, these frameworks must be implemented such that the planning and execution representations of the world can be mapped one to another.

Dvorak et al. (2014) introduce the Flexible Acting and Planning Environment. This framework combines plan-space planning approaches with simple temporal networks and hierarchical decomposition rules allowing plan repair, extension, and replanning, while being able to check and keep up to date temporal relations and constrains consumption. This work also contains a dispatcher that calls for each planned action a set of skills and tracks their evolution over time. The skills are functions that can process input data and compute commands for the actuators.

Buksz et al. (2018) propose a two-level hierarchical planning approach to achieve long autonomy in scenarios with underwater vehicles. In the first step of their approach, the set of all goals for a mission is split in n clusters. For each cluster, a plan is generated. The plan generation for a set of sub-goals is called the tactical planning. The tactical plans are then used on the higher-level planning, called the strategic planning. The initial and goal states of the tactical plans are directly mapped to states on the strategic level. However, they do not consider any recovery procedures in case of failures on any of the planning levels.

To enforce the consistency between the planning and execution representations, Cashmore et al. (2019) proposes the concept of bail-out actions for planned actions that are not executed as expected. The bail-out actions can be seen as recovery procedures on the execution level (e.g., of a navigation action) that force the system to a state from which planning can continue. A similar concept for recovery procedures for Pentri-Net plans is presented in (Iocchi et al. 2016).

Bezrucav and Corves (2020) introduced the Three-Level Planning Approach for scenarios with collaborating humans and robots. On the first level, all goals of the scenario planning problem are clustered. On the second level, temporal planning is deployed for each cluster of goals, while on the third level, FSMs map each temporal action to low-level commands (e.g., calls to a path planner). Their work also contains recovery approaches on the third level in case that errors occur during the execution of the planned actions. A failure in the execution of one of the low-level commands implies an error in one state of a FSM. To guarantee that in case of a failure any executor (e.g. FSM) brings the system to a state from which replanning on the second level is possible, all executors have one special characteristic: they can end in their start state, their goal state, or in an error state. The first two states allow the mapping of a third-level state to a second level state (e.g. the temporal planning state) from which replanning is possible. However, their recovery procedure is limited by the specific end states allowed for the executors (e.g. FSMs).

All works presented in this section only address parts of a generic approach that identifies the valid states for the given initial state and planning domain and enables replanning.

Background

Automated planning with PDDL (Fox and Long 2003), or related formalisms, discretises the events in the continuous time signal. For instance, durative actions may have effects at the beginning or ending of the execution, while classical planning assumes all effects at the end. In reality, though, effects will happen along the execution of the action. Therefore, in case of an execution failure, some effects may not be set as planned. In that case, the system can result in a state that is not consistent with the states expected by the planner. When this happens, the planner may not be able to find any solution when replanning to the same or new recovery goals.

An example of this is that of a robot moving. A navigation action is usually modelled (in temporal planning) with a precondition of the robot being in a location, an initial effect of the robot not being in that location, and an ending effect of the robot being in a new location. However, planning-wise, the robot is nowhere during the action execution, as it is moving between the two places. Thus, if the action fails after the robot leaves the initial location and before it reaches the goal location, the system ends in an *improper* state. If the planning domain contains only the navigation action as described above, the planner will not find a new plan, as moving would require for the robot to be somewhere. But the robot is nowhere in the updated planning state after the failed execution, although it has not disappeared in the real world. This is an example that clearly shows the limitations of the model, where some planning states may not represent reality. We consider those states to not be *proper states*.

The notion of state invariants will be used to define and compute when a state is *proper*. Following Rintanen (2000) and Lipovetzky, Muise, and Geffner (2016), we define state invariants in the context of automated planning formulated in PDDL as follows.

Definition 1. *A state invariant is a formula that is true in all states that can be reached from a given initial state, given a set of operators from a planning domain.*

State invariants can be statically inferred from a planning domain and an initial state, for instance, by using the Type Inference Module (TIM) introduced by Fox and Long (1998). An example of such an invariant is the *at* predicate from a navigation planning problem. In all expected states of the planning domain, if no actions are executing, the invariant $i_1 = |at(robot)| = 1$ holds. This means that the *robot* is exactly at one location in each of these states. These invariants are used in the following section to introduce the concept of proper states.

Proper States and Execution Mappings

With the focus set on planning, we formalize the concept of proper states and their mappings to different levels of execution granularities. We define a planning problem as:

Definition 2. *A planning problem* Π *is a 4-tuple* $\Pi = \langle S, A, s_0, g\rangle$ *where* S *is a finite set of states (both boolean predicates and numerical fluents),* A *is a finite set of actions where every action* $a \in A$ *has some preconditions* a_{pre} *and effects* $a_{eff} = \langle a_{add}, a_{del}\rangle$, $s_0 \in S$ *is the initial state, and* g *is a set of goals. A solution to the planning problem is a plan* π *containing a set of grounded actions* $a_i \in A$ *that bring the system from* s_0 *to a goal state in which the goals from* g *hold.*

Given a set of invariants I for a planning problem Π, we define a proper state as follows.

Definition 3. *A state* $s_p \in S$ *in a planning problem* $\Pi = \langle S, A, s_0, g\rangle$ *is a proper state if it satisfies all invariants in* I *that are also satisfied in the initial state* s_0.

In this context, a *improper state* is a state in which some of the invariants are violated. From such a state, planning could be performed, but either no plans would be found or any found plans would not correspond to the logic described in the planning problem. In contrast, a *proper state* is a state that still fits in the logic described in the planning problem. Therefore, the execution procedures of each planned action must guarantee that the system always ends[1] in a proper state to increase the chances of a successful replanning.

Plan executors tend to be layered, with every action in the plan being executed by a lower-level representation that at the same time may be an abstraction of an even lower-level representation. Such underlying representations may also be state-based. As an example, the executor of a planning action may be a Finite State Machine (FSM), or even another planning problem that tackles the specific action of top-level planning problem (Buksz et al. 2018).

[1]Ending as in stops after a failure, before replanning.

Given the lower-level execution representations of the actions, we define a mapping between states in the different representational layers.

Definition 4. *Given a planning problem* $\Pi = \langle S, A, s_0, g\rangle$ *with a set of proper states* $S^+ \subseteq S$, *and an underlying state-based execution representation consisting of a state space* U, *a proper mapping is a function* $M : U \rightarrow \{S^+, \perp\}$ *that maps execution states to planning proper states, or* $\perp$ *if the execution state corresponds to a improper state.*

Without loss of generality, we focus on the case of having an FSM representing the underlying action execution.

Definition 5. *A deterministic Finite State Machine (FSM) is a 5-tuple* $\langle \Sigma, \bar{S}, \bar{s}_0, \delta, F\rangle$ *where* Σ *is the input alphabet (finite set of symbols),* $\bar{S}$ *is the finite set of states,* $\bar{s}_0 \in \bar{S}$ *is an initial state,* $\delta : \bar{S} \times \Sigma \rightarrow \bar{S}$ *is a state-transition function, and* $F \subseteq \bar{S}$ *is the set of final states.*

We extend the FSM definition by labeling the transitions with partial effects of the planning action they represent.

Definition 6. *A planning executor Finite State Machine (eFSM)* $\langle \Sigma_a, \bar{S}_a, \bar{s}_{a_0}, \delta_a, F_a, E_a\rangle$ *is an FSM that executes some planning action* a, *and whose transitions are labeled by a subset of the action's effects* a_{eff}. *The transition function for an eFSM is then defined as* $\delta_a : \bar{S}_a \times \Sigma_a \rightarrow \bar{S}_a \times E_a$, *where* $E_a = \{\langle p, d\rangle \mid p \in \mathcal{P}(a_{add}) \wedge d \in \mathcal{P}(a_{del})\}$, *where* $\mathcal{P}(\cdot)$ *denotes the powerset.*

Zero or more effects of action a may be assigned to each transition, as described in the extended state transition function δ_a. When an eFSM transition from $\bar{s}_{a,i} \in \bar{S}$ to $\bar{s}_{a,j} \in \bar{S}$ is not labeled with any effect from a, the resulting state $\bar{s}_{a,j} \in \bar{S}_a$ will have the same mapping as $\bar{s}_{a,i} \in \bar{S}_a$. This is because the granularity of the different state space representations is different. Therefore, while the eFSM state may have transition, the planning state may have not changed. In this context, the set U in Definition 4 is equivalent to an eFSM state space $\bar{S}_a$.

Analysing Execution State Spaces

In this section, we formulate a theorem to show the applications of proper states and eFSMs in integrated planning and execution approaches for real world scenarios.

Theorem 1. *If the eFSM* $\langle \Sigma_a, \bar{S}_a, \bar{s}_{a_0}, \delta_a, F_a, E_a\rangle$ *of any action* $a \in A$ *of a planning problem* $\Pi = \langle S, A, s_0, g\rangle$ *has final states* $\bar{s}_a \in F_a$ *for which the mapping* $M(\bar{s}_a) = s, s \in S^+$ *exists, then the execution of the plan is guaranteed to always end in a proper state.*

Proof. Each plan π for a planning problem Π has a set of grounded actions A_f after which no further action is planned. If the plan is sequential, the set A_f has only one action. If the plan contains concurrent actions, the set A_f may have more than one element. If the eFSMs of all actions in the planning model $a \in A$ end in execution states that map to proper states, then all actions $a_f \in A_f$ must end in execution states that are mapped to proper states $s_i \in S^+$. Given that the invariants I hold in each s_i, they will also hold in the state reached when all actions $a_f \in A_f$ have finished. Thus, the final execution state will also be a proper state regardless of the execution outcome (success or failure). □

The execution of the plan can end not only when every action is carried out as expected, but also when an execution error occurs. In this case, all other actions that were executing at the time point when the error occurred should continue as planned.

The proposed formalizations provide a framework for increased execution robustness, allowing executors to be statically checked. In the eFSM executor case, robustness can be improved by ensuring all eFSM's final states have a defined mapping to a proper planning state. Our formalization can be used to check if final states have this defined mapping and to prompt the user when final eFSM states could end up in improper planning states so that they add recovery transitions.

Application Example and Discussion

In this section, we present our approach on a realistic planning problem Π_{rob} for a robotic application. The PDDL planning domain for Π_{rob} contains three actions: *navigate*, *grasp*, and *place*, with corresponding predicates and type definitions. The definitions of actions *navigate* and *grasp* are presented in Figure 2.

```
(:durative-action navigate
  :parameters (?v - robot ?from
  ?to - waypoint)
  :duration(= ?duration 3)
  :condition (and
    (at start (robot_at ?v ?from)))
  :effect (and
    (at start (not (robot_at ?v ?from)))
    (at end (robot_at ?v ?to))))
(:durative-action grasp
  :parameters (?v - robot ?p - object
  ?place - waypoint)
  :duration(= ?duration 2)
  :condition (and
    (over all (robot_at ?v ?place))
    (at start (robot_at ?v ?place))
    (at end (robot_at ?v ?place))
    (at start (object_at ?p ?place))
    (at start (stowed ?v))
    (at start (empty_gripper ?v)))
  :effect (and
    (at start (not (object_at ?p ?place)))
    (at start (not (empty_gripper ?v)))
    (at start (not (stowed ?v)))
    (at end (stowed ?v))
    (at end (object_at ?p ?v))))
```

Figure 2: Definition of the durative-actions from the planning domain of Π_{rob}
.

The PDDL planning problem for Π_{rob} contains the initialization of the robot, the object, and five waypoints. Beside these, the PDDL planning problem also contains the initial and goal states (see Figure 3).

```
(:init
 (robot_at robot wp1)
 (object_at obj wp3)
 (stowed robot)
 (empty_gripper robot))
(:goal (and
 (object_at obj wp5)
 (robot_at robot wp2) ))
```

Figure 3: Initial state and goals for Π_{rob}.

Following, we perform a TIM (Fox and Long 1998) analysis on the planning problem Π_{rob} to determine the state invariants I. Based on these state invariants, we create eFSMs that will make the execution more robust, and will not allow the system to drift to an improper state. Figure 4 shows an excerpt of TIM's output that encode the invariants I.

```
{} => {robot_at_0 } -> {navigate_0 } → i1
{} => {navigate_0 } -> {robot_at_0 } → i1
{} => {object_at_0 } -> {grasp_1 } → i2
{} => {grasp_1 } -> {object_at_0 } → i2
{} => {object_at_0 } -> {place_1 } → i3
{} => {place_1 } -> {object_at_0 } → i3
{robot_at_0 } =>
 {stowed_0 ...}->{grasp_0} → i4
{robot_at_0 } =>
 {grasp_0}->{stowed_0 ...} → i4
```

Figure 4: Snippets of the output delivered by TIM for the planning problem Π_{rob}.

The first line from Figure 4 can be interpreted as follows: when the *robot_at* predicate holds for its first parameter (0th in TIM's output), the *navigate* action can be applied and its first parameter is the same as the first parameter of the *robot_at* predicate (e.g., *robot*). The first and the second line can be interpreted together as: when the *robot_at* predicate holds for its first parameter and the action *navigate* is applied, in the obtained state, the predicate *robot_at* should also hold for the same parameter. These two lines represent the first invariant i_1 that says that the predicate *robot_at* should hold for a parameter of type *robot* throughout all the execution of the *navigate* action. Invariants $i_2 - i_4$ are determined in a similar manner. They imply that the predicate *object_at* must hold for an *object* all the time during the execution of the *grasp* and *place* actions (i_2 and i_3), while the arm must also be *stowed* during the *grasp* action (i_4). If all these invariants hold when no actions are executing, then the state remains *proper* and replanning is more likely to succeed. To ensure this, specific eFSM must be created for the three actions.

The *navigate* action is implemented on the execution level by an eFSM with three states (see Figure 5). The initial state $\bar{s}_{nav_1}$ is the start state where the robot is at $wp{-}from$. $\bar{s}_{nav_2}$ is the state obtained if the navigator (e.g. implemented with a path planner) is successful and the agent reaches $wp - to$. The transition connecting these two states has two effects. One effect deletes the fact that the agent is at $wp - from$,

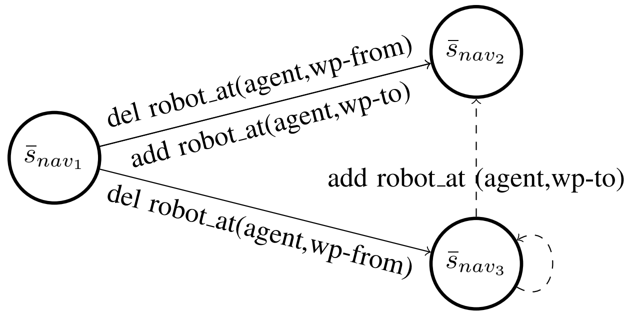
$\bar{s}_{nav_1}$
$\bar{s}_{nav_2}$
$\bar{s}_{nav_3}$
del robot_at(agent,wp-from)
add robot_at(agent,wp-to)
add robot_at (agent,wp-to)
del robot_at(agent,wp-from)

Figure 5: The original eFSM representation and its extension with the dashed transitions to guarantee that all its final states can be mapped to planning proper states.

while the second effect adds the information that the agent is at $wp - to$. $\bar{s}_{nav_3}$ is the third execution state that is reached when the underlying navigation action (e.g. a path planner) fails. The transition connecting $\bar{s}_{nav_1}$ to $\bar{s}_{nav_3}$ has attached only an effect that deletes the information of the agent being a $wp - from$.

With the original eFSM, the execution of the *navigate* action can end in state $\bar{s}_{nav_3}$ for which invariant i_1 does not hold anymore. Thus, the system would reach an improper state after failure. To ensure that the *navigate* eFSM can end only in proper states, this eFSM must be extended with two further transitions. One transition is added between $\bar{s}_{nav_3}$ and $\bar{s}_{nav_2}$ and it describes the case when a newly triggered navigation command finishes successfully. In this case, the system lands in state $\bar{s}_{nav_2}$ and the *robot_at* effect must be added. If the second navigation command does not execute as expected, the system remains in the improper state $\bar{s}_{nav_3}$, as described by the second dashed transitions. With the two new transitions, the navigate eFSM has only one end-state, namely $\bar{s}_{nav_2}$ that maps to a planning proper state. Figure 5 shows the original representation of the eFSM and its extensions (e.g. dashed transitions).

We now show another example using the *grasp* action. Here, invariants i_2 and i_4 should hold. Figure 6 presents the execution of action on the planning level with the evolution of the grounded predicates in the planning state. These grounded predicates change when the corresponding *add* and *delete* effects from the *grasp* action are applied. On the execution level, the *grasp* action is first implemented with an eFSM with four states (in blue):

- $\bar{s}_{grasp_0}$ is the initial state
- $\bar{s}_{grasp_1}$ is the state reached after the object is approached
- $\bar{s}_{grasp_2}$ is the state reached after the gripper is closed
- $\bar{s}_{grasp_3}$ is the state reached after the arm is stowed.

The action effects from the planning level are applied on the transitions between these eFSM states, as shown in the lower level of Figure 6.

If the eFSM *approach object* action fails, the *robot* can not further grasp the *object*, thus, it can not continue with the initial plan. The system would remain in the execution state $\bar{s}_{grasp_1}$, where the *object* is nowhere, the *arm* is not stowed, and a replanning is required. In order to prevent the system reaching an improper state, the status of the invariants must be checked before replanning. Although the *grasp* action has completed on the planning level, the *object_at* predicate for the *obj* and the *stowed_arm* predicate were deleted from the state during the action execution and were not set up again due to the failure. Thus, invariants i_2 and i_4 do not hold in the resulting state, which means $\bar{s}_{grasp_1}$ is improper. To avoid such behaviours, the eFSM for the *grasp* action is extended with two further states (represented in green in Figure 6). $\bar{s}_{grasp_4}$ is reached from $\bar{s}_{grasp_1}$ after an object detection action is executed, while $\bar{s}_{grasp_5}$ is reached from $\bar{s}_{grasp_4}$ after a stowing arm action is carried out. The transitions between the new introduced states also reset corresponding predicates, and a transition between states $\bar{s}_{grasp_5}$ and $\bar{s}_{grasp_0}$ is also added. With the new eFSM, the execution of the *grasp* action becomes more robust. In case that approaching the object action fails, a recovery procedure is now available that updates the state of the world on the planning level, such that a proper state (e.g., $\bar{s}_{grasp_0}$) can be reached, where the predicates *object_at obj* and *stowed_arm* hold. Therefore, invariants i_2 and i_4 also hold, which means that the state is proper and replanning is possible. However, the extended eFSM is not yet complete. If the object is not detected or the arm can not be stowed after a failure of the approach object action, the system may still result in improper states.

Concluding, the presented approach can be used to improve the execution robustness of any planning problem. This improvement is achieved by checking for the executors of all actions $a \in A$ of Π, if they satisfy the invariants and are able to repair the *improper states* of the planner's model that do not accurately match a real ground truth. Here, we propose to use the concept of proper states and invariant checks as an assisting tool (similar to the automatic plan validation tool VAL (Howey, Long, and Fox 2004)) to check the completeness of the actions' execution description, and to devise potential executor errors.

Conclusions and Future Work

This paper presents a formalization towards making integrated planning and execution approaches for embodied agents more robust against execution failures. The main contributions are the definition of the proper states on the planning level and the mapping between states from the planning and the execution levels. The proposed formalization provides a framework to check if execution representations may result in an artificial dead-end from which planning cannot continue, paving the way towards automatic recovery behaviors of such kinds of failures. Here, we have proposed manual extensions of the eFSMs to recover from improper states.

In future work, we will look into the automatic extension of eFSMs such that all their final states can be mapped to proper planning states, ensuring the ability of continue with planning when execution fails. This will imply determining the missing predicates to reach a planning proper states from a failed state and the generation of appropriate transitions that recover the planning state.

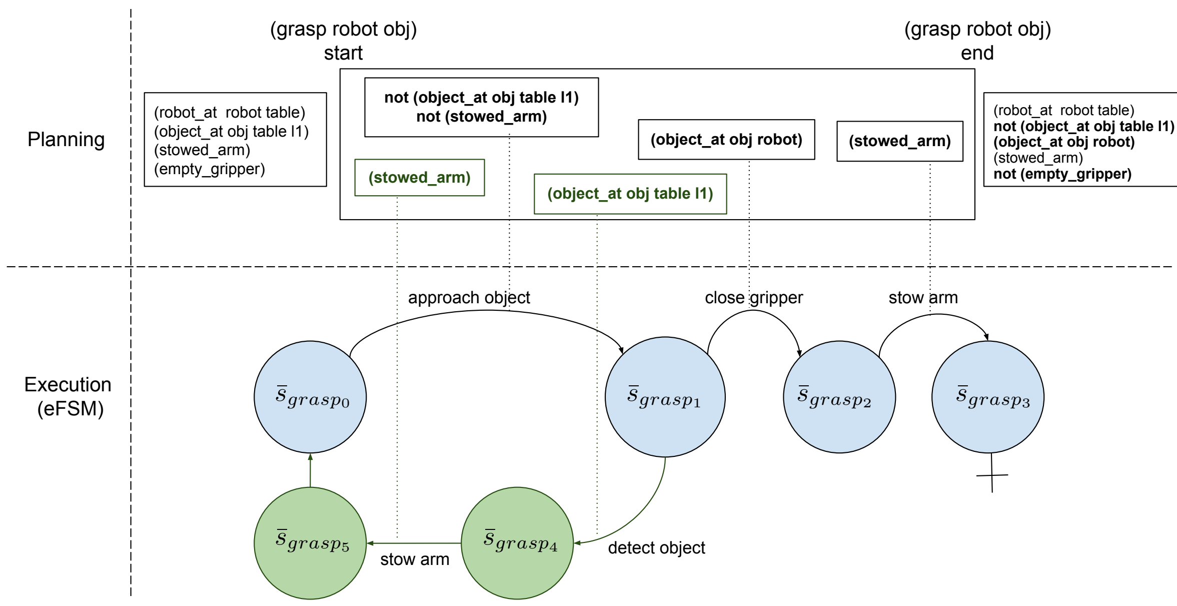

Figure 6: Example of mapping between planning and execution levels in the context of a grasping action. We include states which would not be proper and potential recovery behaviors.

Acknowledgements

This work has been partially supported by the European Union's Horizon 2020 project Sharework (grant agreement No. 820807). Gerard Canal is supported by the Royal Academy of Engineering and the Office of the Chief Science Adviser for National Security under the UK Intelligence Community Postdoctoral Research Fellowship programme.

References

Bezrucav, S.-O.; and Corves, B. 2020. Improved AI Planning for Cooperating Teams of Humans and Robots. In *Workshop on Planning and Robotics (PlanRob) at the 30th International Conference on Automated Planning and Scheduling*. 10.5281/zenodo.4286242.

Buksz, D.; Cashmore, M.; Krarup, B.; Magazzeni, D.; and Ridder, B. 2018. Strategic-Tactical Planning for Autonomous Underwater Vehicles over Long Horizons. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3565–3572. IEEE. ISBN 978-1-5386-8094-0.

Cashmore, M.; Coles, A.; Cserna, B.; Karpas, E.; Magazzeni, D.; and Ruml, W. 2019. Replanning for Situated Robots. In Benton, J.; Lipovetzky, N.; Onaindia, E.; Smith, D. E.; and Srivastava, S., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling*, 665–673. AAAI Press.

Cushing, W.; Benton, J.; and Kambhampati, S. 2008. Replanning as a Deliberative Re-selection of Objectives.

Dvorak, F.; Bartak, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Planning and Acting with Temporal and Hierarchical Decomposition Models. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, 115–121. IEEE. ISBN 978-1-4799-6572-4.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan Stability: Replanning Versus Plan Repair. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, 212–221. AAAI Press. ISBN 978-1-57735-270-9.

Fox, M.; and Long, D. 1998. The Automatic Inference of State Invariants in TIM. *Journal of Artificial Intelliigence Research*, 9: 367–421.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research (JAIR)*, 20: 61–124.

Gerevini, A.; and Serina, I. 2000. Fast Plan Adaptation through Planning Graphs: Local and Systematic Search Techniques. In Chien, S. A.; Kambhampati, S.; and Knoblock, C. A., eds., *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, 112–121. International Conference on Artificial Intelligence Planning and Scheduling, AAAI Press. ISBN 978-1-57735-111-5.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence*, 294–301. ISBN 0-7695-2236-X.

Iocchi, L.; Jeanpierre, L.; Lazaro, M.; and Mouaddib, A. 2016. A Practical Framework for Robust Decision-Theoretic Planning and Execution for Service Robots. In Coles, A.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling*, 486–494. AAAI Press. ISBN 978-1-57735-757-5.

Lipovetzky, N.; Muise, C.; and Geffner, H. 2016. Traps, Invariants, and Dead-Ends. In Coles, A.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling*. AAAI Press. ISBN 978-1-57735-757-5.

Nebel, B.; and Koehler, J. 1995. Plan reuse versus plan generation: a theoretical and empirical analysis. 76(1-2): 427–454. PII: 000437029400082C.

Rintanen, J. 2000. An Iterative Algorithm for Synthesizing Invariants. In *Seventeenth National Conference on Artificial Intelligence (AAAI-2000), Twelfth Innovative Applications of Artificial Intelligence Conference (IAAI-2000)*, 806–811. American Association for Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference, AAAI Press and MIT Press. ISBN 978-0-262-51112-4.

van der Krogt, R.; and de Weerdt, M. 2005. Plan Repair as an Extension of Planning. In Myers, K.; Rajan, K.; and Biundo, S., eds., *Proceedings, the fifteenth international conference on automated planning and scheduling*. AAAI Press. ISBN 978-1-57735-220-4.