PELEA: a Domain-Independent Architecture for Planning, Execution and Learning

[1] César Guzmán, [2] Vidal Alcázar, [3] David Prior
[1] Eva Onaindia, [2] Daniel Borrajo, [3] Juan Fdez-Olivares, [2] Ezequiel Quintero
[1] Universitat Politècnica de València, Camino de Vera, s/n, 46022 Valencia, Spain
[2] Universidad Carlos III de Madrid, Av. de la Universidad, 30, 28911 Leganés, Spain
[3] Universidad de Granada, Av. Hospicio, s/n, 18010 Granada, Spain
cguzman@dsic.upv.es, valcazar@inf.uc3m.es, dprior@decsai.ugr.es
onaindia@dsic.upv.es, dborrajo@ia.uc3m.es, faro@decsai.ugr.es and equinter@inf.uc3m.es

Abstract

One of the current limitations for large-scale use of planning technology in real world applications is the lack of software platforms to integrate the full spectrum of planning-related technologies: sensing, planning, executing, monitoring, re-planning and learning from past experiences. In this paper, we present PELEA, a domain-independent online planning architecture that includes state-of-the-art components for performing a wide range of (meta-)planning tasks, such as learning of control knowledge or low-level planning among many others. PELEA is conceived as a general-purpose architecture suitable for problems ranging from robotics to emergency management. PELEA is also intended to provide a rapid prototyping life-cycle for building planning applications and support planning practitioners with a highly-configurable tool.

Introduction

Automated Planning (AP) has been successfully applied to different real-world problems, such as space (Ai-Chang et al. 2004), robot control (McGann et al. 2009), or fire extinction (Fdez-Olivares et al. 2006) among many others. The process of developing a final application is an "ad-hoc" manual process that requires expertise and techniques from several fields as well as a careful definition of the underlying architecture. Most applications rely on architectures that include the functionalities required for a continuous planning, namely sensing the state, generating the problem at hand, planning , executing the plan, monitoring the execution for failures, etc. These applications are also based on replanning when needed, and, possibly, learning from the interaction to generate better models or control knowledge to improve search. However, in most applications, specifically tailored software had to be developed for the domain at hand, which usually lacks generality and reuse possibilities.

There have been some attempts, though, to design generic architectures used for different purposes. Examples can be found in space and robotics applications of platforms as Mapgen (Ai-Chang et al. 2004), APSI (Cesta et al. 2009), PRS (Georgeff and Lansky 1987), or IxTeT (Ghallab and Laruelle 1994). However, these platforms have been designed for specific problems and techniques, as timeline-based planning (Ai-Chang et al. 2004; Cesta et al. 2009; Ghallab and Laruelle 1994), hierarchical planning (Fdez-Olivares et al. 2006), or reactive controllers (Georgeff and Lansky 1987).

In this paper, we present PELEA, a domain-independent, component-based architecture able to perform planning, execution, monitoring. repairing and learning in an integrated way, in the context of PDDL-based and HTN-based planning (Alcázar et al. 2010). PELEA follows a continuous planning approach, i.e. an ongoing and dynamic process in which planning and execution are interleaved but, unlike other approaches (Myers 1999; Chien et al. 2000), it allows planning engineers to easily generate new applications by reusing and modifying the components as well as a high flexibility to compare different techniques for each module or even incorporate one's own techniques.

One particular application domain suitable for testing a continuous planning approach is *robotics* as it provides the kind of plan generation and replanning capabilities required for situated agents in highly dynamic environments. We use the general-purpose, highly-configurable architecture PELEA as an autonomous mobile robot control system. PELEA allows controlling the execution of a given task independently of the robot control platform and devices, monitoring the correct plan execution, resolving uncertainty by replanning when needed, and learning additional knowledge. To test PELEA as a robot control system we worked with the *Rovers* domain from the International Planning Competition (IPC[1]). This domain is a simplified version of the planning tasks performed by the Mars Rovers.

This paper is organized as follows. First, we present an overview of PELEA architecture. The next two sections relate the high-level and low-level planning in PELEA, respectively. The following sections describe the learning module and the goals and metrics module. Afterwards, a case study showing the features of PELEA is shown. Finally, the last section presents our conclusions.

[1] IPC: http://ipc.icaps-conference.org/

Overview of PELEA Architecture

A full description of PELEA architecture can be found in (Alcázar et al. 2010). Here, we sketch the components and main functionalities of PELEA.

PELEA architecture includes components that allow the applications to dynamically integrate planning, execution, monitoring, replanning and learning techniques. PELEA provides two main types of reasoning: high-level (mostly deliberative) and low-level (mostly reactive). This is common to most robotics applications and reflects the separation between a reactive component and a deliberative component. However, in our architecture, these are simply two planning levels. This offers two main advantages: both levels can be easily adapted to the requirements of the agent; and the differentiation allows the agent replanning at either level, which grants a greater degree of flexibility when recovering from failed executions.

Figure 1 shows a screenshot of PELEA's web interface and the current version of the architecture along with the integration of the modules. As we can see, PELEA is composed of eight modules that exchange a set of Knowledge Items (KI) during the reasoning and execution steps. We have chosen to use XML within the architecture to represent those KIs, which are: (1) *stateL*, low-level state composed of the sensory information; (2) *stateH*, abstracted high-level state translated from stateL as an aggregation or a generalization of low level information; (3) *goals*: the set of high-level goals to be achieved by the architecture; (4) *metrics*, metrics that will be used in the high-level planning process; (5) *planH*, high-level plan generated with any state-of-the-art high-level planner (this is a configurable parameter in PELEA); actions in planH can also be the goals for the low-level planner (in case we want the low-level planner to act as a dynamic translation mechanism for high-level actions); (6) *planL*, low-level plan as a set of operational actions resulting from the low-level planning that are directly executable in the environment; (7) *domainH*, definition of actions for the high-level planning; (8) *domainL*, definition of behaviors (skills) for the low-level planning learning examples; (9) *heuristics*, in different forms (control rules, policies, cases, macro-actions, etc.) allow the planner to improve the efficiency in solving future planning episodes; and (10) *info monitor*, meta knowledge on the plan that helps perform the plan monitoring (for instance, the generation time of a literal).

The high-level knowledge describes general information, actions in terms of its preconditions and effects, and typically represents an abstraction of the real problem. High-level knowledge is concerned with the description of the high-level domain, problems, goals and metrics, and they are required for the purpose of planning sequences of actions, and for the modifications of these sequences (repair or replanning). We use PDDL to represent this information. However, high-level knowledge descriptions are rarely directly executable, if ever, and they must be complemented by the low-level knowledge, which specifies how the operations are actually performed in terms of continuous change, sensors and actuators. Low-level knowledge describes the more basic actions in the simulated world, and it is typically concerned with specific rather than general functions, and how they operate. Now, the main components of PELEA are described.

Execution Module. The starting point of PELEA is the initialization of the Execution Module to capture the current problem state (*stateL*). This module also receives a high/low-level domain, and a problem. The Execution Module keeps only the static part of the initial state, given that the dynamic part, *stateL*, is read from the environment through sensors. The environment is either a hardware device, a software application, a software simulator, or a user. The Execution Module is in charge of receiving the new *stateL* and sending out the low-level actions (*planL*) that have to be executed at each step to the actuators.

Monitoring Module. Both the problem and the domain definitions and, optionally, a metric, are sent to the Monitoring Module to obtain a high-level plan (*planH*). Then, *planH* is translated into a low-level plan (*planL*) whose actions are finally sent to the Execution Module. In PELEA, it is not necessary to work at the two knowledge levels. One can just work at the high-level, so that converting knowledge from high-level into low-level with the LowToHigh module or using the Low-level planner module are not required. The Monitoring Module calls the Decision Support, which in turn calls the High-level replanner, to obtain *planH*. If the low level is being used, *planH* is converted into *planL*; otherwise, (some) actions in *planH* are directly sent to the Execution Module. Once the actions are executed, the Monitoring Module receives the necessary knowledge (current state, problem and domain) from the Execution Module and it starts the plan monitoring process. The first step is to check whether the problem goals are already achieved in the received state (*goalsL* and *goalsH* in case we are dealing with the two processes). If so, the plan execution finishes; otherwise, the Monitoring Module checks whether the received state matches the expected state or not and determines the existence or lack of a plan failure.

Low-level planner. The Monitoring Module, with the help of the Low-level planner module, generates a set of executable low-level actions (*planL*). An example of low-level knowledge would be "the coordinates of a robot" or "degrees of motion of a robot arm". If the Low-level planner module is not used, the Monitoring Module assumes that actions in *planH* are executable, and they are directly sent to the Execution Module.

Decision Support Module. It selects the variables to be observed by the Monitoring Module during the plan monitoring, and takes the decision of repairing or re-planning through an Anytime Plan-Adaptation approach (Garrido, Guzman, and Onaindia 2010) when the Monitoring detects a failure in the plan monitoring. It also communicates the Monitoring Module with the High-level replanner Module and retrieves training instances from the execution and the plans to be sent to the Learning module.

High-level replanner. It receives a problem and a high-level domain (*domainH*) and generates a high-level plan (*planH*). This module is also invoked when the Decision Support has to fix (repair/replan) a plan. In this latter case, the initial state of the problem will be the current observed

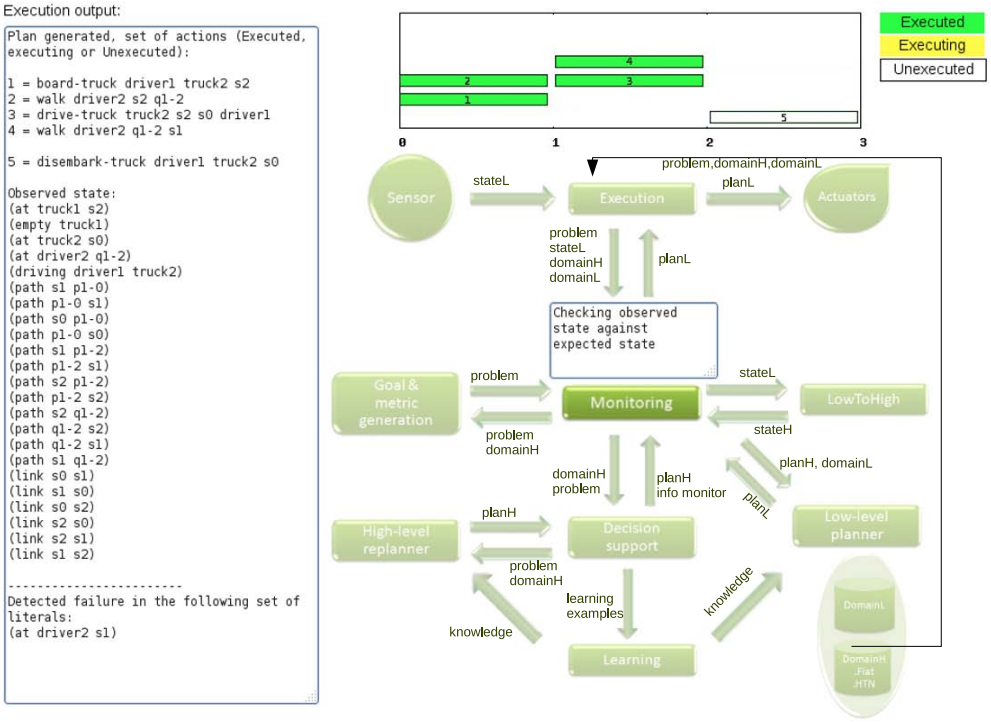

Figure 1: Screenshot of PELEA's web interface showing the architecture of the system. It shows the execution of a simple problem in the Driverlog domain.

state. Several planners have been successfully used for this module: LPG-TD (Gerevini, Saetti, and Serina 2003), SGPLAN (Hsu et al. 2007), CRIKEY (Coles et al. 2009) and TFD (Eyerich, Mattmller, and Rger 2009).

Learning Module. It infers knowledge from a training set sent by the Decision support module. The knowledge can be used either to modify the domain planning model or to improve the planning process (heuristics). Apart from the different levels of reasoning, PELEA can also learn from past executions and reason about the current problem to improve its efficiency.

The components run as separate processes and communicate through sockets. The inputs are defined by either the PDDL or HTN domain/problem specification at the high level. The knowledge exchanged among components follows the domain-independence principle with domain-independent APIs (through XML). Here lies the generality of PELEA; one can exchange a component and PELEA will continue working as it is, maintaining the XML APIs and their semantics, which are the standard ones in planning: actions, goals, states and plans.

Plan Monitoring and Decision Support

The *info monitor* parameter, provided by the Decision Support Module, comprises the information that needs to be monitored to guarantee a successful plan execution. Specifically, it includes: i) the variables to be monitored, i.e. those that are directly related to the plan, ii) the time at which the variable is generated, and the earliest and latest time at which the variable will be used, respectively; and iii) the value range for each variable, denoting the set of correct values that the variables can take on. The Decision Support Module computes the variables to be monitored through an extension of the goal regression method proposed in (Fritz and McIlraith 2007), which is inspired by the mechanism used in triangle table defined in (Fikes, Hart, and Nilsson 1972). This mechanism is only used so far to monitor the high-level information.

The Monitoring Module receives *planH* and the *info monitor* parameter and sends a set of executable actions from *planH* at a time instant t to the Execution. For example, in figure 2, at $t = 0.0003$ actions a_1 and a_2 can be executed in parallel. The Monitoring sends these two actions to the Execution and requests the execution state at the time in which variables are generated and used, most typically at the end of the execution of the actions. In this example, the Monitoring, with the help of the Execution, senses the dynamic state variables from the environment at $t = 2.0003$.

Once the information of the observed state is received by the Monitoring at time t, it checks the values of the variables are within the value range specified in *info monitor* parameter. If so, the Monitoring continues with the plan execution, sending the next set of actions to the Execution (action a_3 in figure 2). Otherwise, a discrepancy between the expected

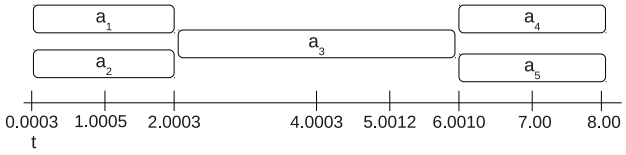

Figure 2: Example of a parallel plan.

and the observed state is found, in which case the anomaly is reported to the Decision Support, which determines whether the discrepancy is relevant to the plan execution or not. At this point, if reactivity is needed, the low-level planner is invoked to find the most immediate actions as this module typically stores predefined policies or courses of actions designed to reach particular goals. In the anomaly entails the plan is no longer executable, the Decision Support is called to take a decision between repairing the current plan or replanning from scratch.

The decision between repairing or replanning is done via the application of a regressed goal-state heuristic (Garrido, Guzman, and Onaindia 2010). A regressed goal state is a tuple of the form $GS = \langle L, t \rangle$ where L is the set of atoms, i.e. values of the state variables, and t is the time of GS, which usually coincides with the start time of one action (sequential planning) or more than one action (parallel planning). The heuristic estimates the best GS according to parameters as the cost or stability of the estimated plan. Then a new problem from S_0 to the selected regressed goal state is generated and the planner is invoked. Note that the first GS is the one from which the whole original plan can be reused; the subsequent goal states represent reachable states from which to reuse ever decreasing parts of the original plan; and the final GS entails no reuse of the plan at all.

Temporal plan monitoring

When monitoring a temporal plan, the classical definition of a regressed goal state, GS, to guarantee the executability of the plan tail from t is no longer sufficient. A *temporal regressed goal state* has now to include actions concurrent with the state variables at t, and the timing of those actions relative to the variables (Haslum 2006). That is, when computing $GS = \langle L, t \rangle$, the algorithm must take into account the actions that are being executed at time t. We will call these actions *ongoing actions*.

A temporal regressed goal state is a tuple $GS = \langle L, t, A \rangle$ where L is the values that the variables should take on at t and $A = \{(a_1, \delta_1), \ldots, (a_n, \delta_n)\}$ is the set of ongoing actions, a set of actions a_i with time intervals δ_i, meaning that each action (a_i, δ_i) in A has started δ_i time units earlier. In other words, a plan achieves the temporal regressed state GS iff the plan achieves L at t and schedules action a_i at time $t - \delta_i$ for each $(a_i, \delta_i) \in A$.

For example, consider the temporal regressed goal state $GS_1 = \langle Ls_1, 5.0012, \{(a_3, 3)\} \rangle$ in figure 3 (a sketch of a plan from a rovers problem). Since $t = 5.0012$ is the starting point of the action a_4, the preconditions of a_4 are subgoals (variables) to be achieved by t. Additionally, the action that achieves those conditions must be compatible with the on-

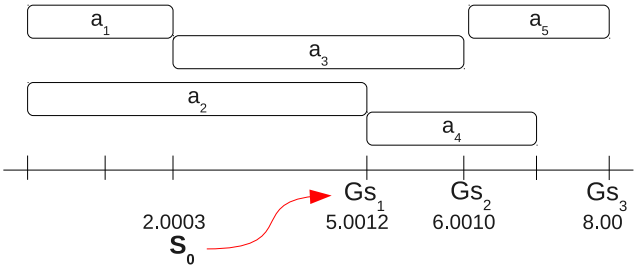

Figure 3: Temporal plan example - *planH* solution.

going action a_3, which starts 3 time units earlier. Ongoing actions are automatically computed by the Decision Support and encoded as *special PDDL actions* in the new planning domain file. Particularly, an ongoing action a_i of a temporal regressed state GS_j has the same specification as the original action except that the `at end` effects are ignored and that the duration is δ_i. The `at start` effects of the original action are included as variables in GS_j.

Analogously to the temporal regressed goal states, there may be actions that are actually being executed at the current time in the observed state (S_0). In this case, the ongoing actions have already been executed for a certain time so it is only necessary to execute them during the remaining time up to the completion of the action. See for example figure 3 where two time units of action a_2 have already been executed at S_0. Now, the current observed state is specified as follows: $S_0 = \langle L, t, A \rangle$, where L is the observed variables, t is the current time and $A = \{(a_1, \sigma_1), \ldots, (a_n, \sigma_n)\}$ is the set of ongoing actions at t, a set of actions a_i with time intervals σ_i, meaning that each action (a_i, σ_i) in A remains σ_i time units to complete. In other words, a plan from S_0 must schedule action a_i at current time t with a duration σ_i for each $(a_i, \sigma_i) \in A$. Ongoing actions of the observed state are encoded as *special PDDL actions* with: i) preconditions equal the `overall` preconditions and `at start` effects of the original action, ii) effects equal the `at end` effects of the original action and iii) a duration of σ_i time units.

Replanning *vs.* Plan Repair The Decision Support is capable of deciding between replanning or repairing in a timely fashion. It uses an algorithm with anytime capabilities whereby a first solution plan is rapidly returned, and the solution quality may improve if the algorithm is allowed to run longer (Garrido, Guzman, and Onaindia 2010). The heuristic takes a balanced response between metric (cost, makespan) and plan stability (part of the original plan that can be reused in the new solution plan) (Fox et al. 2006). Plan stability is one of the principal reasons for claiming the preference of plan repair over the alternative of replanning.

The heuristic estimates an *approximate plan* Π_{replan} (a plan from S_0 to the GS_n discarding the whole original plan), and a plan Π_{repair} (a plan from S_0 to the GS_1 keeping the whole original plan). If $cost(\Pi_{replan}) < cost(\Pi_{repair})$ or $stability(\Pi_{replan}) > stability(\Pi_{repair})$ then replanning is chosen as the preferred option. Otherwise, the algorithm analyzes the cost and stability of the subsequent regressed goal

states ($GS_2, \ldots, GS_{n-1}$) and maintains the best regressed goal state computed so far until time expires. Once a goal state GS_i is selected, the High-level replanner is invoked with the initial state S_0 and goal state GS_i. The returned plan is concatenated with the plan tail of the original plan taking into account the causal links and time constraints.

Closer to the Real World: Low Level

Actions in low-level plans (*planL*) are atomic actions that are executed directly in the environment. The low-level reasoning components are often required to come up with a solution in a short time. In this section, we present in detail the PELEA modules that implement the low level behaviour.

Execution Module

The Execution Module deals with the communication between PELEA and the environment, which can be represented by a simulator, a hardware device (robot), a third-party software, or a user. Currently, the Execution Module works as a wrapper over anything external to PELEA, tackling with low-level details that depend on the kind of environment PELEA is working with. Issues like communication protocols and data formats are responsibility of the Execution Module.

Low-level states (*stateL*) are sent to the Monitoring Module upon request. Similarly, actions in *planL* are sent to the actuators when they are received from the Monitoring. This is usually interleaved, although PELEA may ask asynchronously for *stateL* without sending any action to monitor the execution. Currently, the Execution Module provides integration with the following environments:

- MDPSim: PPDDL (Younes and Littman 2004) simulator employed in the past probabilistic tracks of the International Planning Competition (IPC)[2] which generates states stochastically as actions are received. It works with a probabilistic version of the regular PDDL domains, PPDDL, in which the effects of the actions depend on a series of probabilities.
- In-house temporal probabilistic simulator: we have built a new simulator that is able to work with temporal probabilistic domains. The domain definition is similar to the temporal version of PDDL, augmented with probabilistic effects as in PPDDL.
- Stage/Player (Gerkey, Vaughan, and Howard 2003): freeware platform for robot independent control.
- Microsoft Robotics Studio: a robot independent platform similar to Player.
- Freeware software suite for robotics applications, research and education, which also offers the possibility to simulate various kinds of robots.
- AIlife: open platform for simulating social and emotion oriented games.
- Planning framework designed to simulate and solve real-life logistic problems.

[2]Except for the last one held on 2011.

Low-Level Planner

The Low-Level Planner generates *planL* composed of atomic actions that can be directly executed in the environment. It receives *stateL* and a high-level action (the next one from the current *planH*). This high-level action conforms the low-level goal. The low-level actions of the *planL* generated by the Low-level Planner are then sent to the Execution Module. Thus, high-level plans can be decomposed into several low-level actions, keeping the reasoning at both levels distinct. The Low-level planner does not have to be a regular PDDL-based planner. We have implemented several types of low-level planners:

- HighToLow translators, that simply decompose a high-level action into low-level ones with no reasoning. They can be seen as small programs that take as input a high-level action and the low-level state and generate as output a set of low-level actions
- A policy, either learned or manually created. For instance, based on a states-actions table (as in most reinforcement learning approaches).

Suppose we have a domain in which a robot can move along a building and turn around to change its orientation. A high-level plan may contain the *(move location1 location2)* action as the current one, so it is sent to the Low-level planner. The other input would be *stateL* that includes information on the *x* and *y* coordinates of the robot position and its orientation. Then, the Low-level planner solves the path-planning problem and returns a sequence of atomic actions to be sent to the robot; for instance, the sequence *(advance 10)*, *(turn 45 right)*, *(advance 5)*.

LowToHigh Module

The task of the LowToHigh Module is to translate a *stateL* into a *stateH*. In most cases it is just a mapping function between the low and the high level, although more complex functionalities can also be implemented. The requirement of being able to generate a *stateH* from a *stateL* is justified by the necessity of monitoring at both low and high levels. Thus, replanning and repairing can be performed during the high-level execution as well.

Learning in PELEA

The Learning Module is in charge of inferring knowledge from the training data sets sent by the Decision Support module. So far, machine learning techniques have been used to improve the planning process by learning domain models, low-level policies and heuristics. We will shortly describe them here as a summary of the kinds of techniques that PELEA integrates into its core, since most of them have already been published elsewhere.

Learning domain models

The generation of accurate robot control PDDL or PPDDL models for planning is complex. To alleviate this, machine learning has been used to support model generation. As an example, a relational learning approach was developed as

part of the Learning Module of PELEA to learn more accurate robot-action execution durations than the originally modeled ones (Quintero et al. 2011). PELEA starts with a deterministic version of a robotics domain and then executes actions and observes the results of those executions in terms of the function whose effects we want to learn (e.g. navigate). We used TILDE (Blockeel and Raedt 1998) to learn the duration of actions of the Rovers domain using regression trees. Finally, the duration models were later compiled into the PDDL domain specification. The experiments were performed using a Pioneer robot that traveled through different terrain types, generating the learning examples from the type of terrain and the time it took the robot to move from one waypoint to another.

Learning low-level policies

Also, as part of the Learning Module, domain-specific learning techniques can be used at the low-level. As a proof of concept, a policy made to substitute the low-level planner was inferred using reinforcement learning. This was done for the case study described later, in which training examples of the form $\langle state, action, state, reinforcement \rangle$ were employed to create a state-action table.

Learning heuristics

The Learning Module is also able to learn heuristics for improving the planner's efficiency. Relational decision trees were used again to learn how to generate look-ahead states to improve forward search algorithms (De la Rosa et al. 2011). This was done by creating training examples that take into account the helpful actions of previous heuristic evaluations of states belonging to smaller problems of the same domain. The learning system is domain-independent, but planner-dependent, so in the future we would like to include other planner-independent techniques.

Goals and Metrics Generation

The Goal&metric generation Module is designed to automatically select the new goals and metrics to be used according to the current state of the execution. A common use of this module is for oversubscription problems, where not all goals can be satisfied. This problem is generally solved by choosing some goals and discarding others either online or offline. An algorithm that computes an estimation of the cost of achieving a goal from every other goal was implemented so a set that maximizes the number of goals that will be likely achieved can be easily found (García-Olaya, de la Rosa, and Borrajo 2011).

A Case Study: Rovers domain

In this section, we show an instance of the Rovers domain which replicates the expected behavior of the autonomous explorers sent to Mars by NASA in past experiments. It was simulated both by using Player/Stage (Gerkey, Vaughan, and Howard 2003), and two real robots, two Pioneers P3-DX,

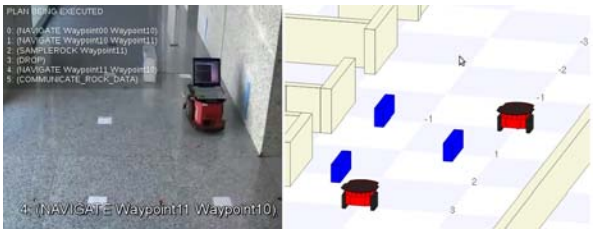

Figure 4: Execution of PELEA: real robot and simulator.

which interacted in a recreated physical space that represented the Mars environment [3].

Both the robot and the simulation were managed using Player/Stage, so there was a single implementation for both cases. The actuators/sensors management was implemented in the Execution Module as a set of basic control skills. The communication was performed using sockets, acting Player/Stage as the server and PELEA as the client. Both the Low-Level Planner and the LowToHigh Module were implemented as an ad-hoc translator adapted for the example.

Description of the case study

We have used the Rovers domain introduced in the IPC in 2002. In this domain, a collection of Rovers navigate on Mars' surface, looking for samples/images data which they should communicate back to Earth. In this case, the STRIPS version of the domain was used for the sake of simplicity. In the executed instance two different Rovers are used, and the goal is to communicate a set of samples taken from rocks in the environment. In the trace example we will refer to one of these two Rovers, named Curiosity. During the execution, both the physical Pioneer robots and the simulator provided by Player/Stage are used. Figure 4 shows the real robot during execution and a screenshot of the simulator at that instant.

Trace of execution

Now, we show the execution of the action `(navigate curiosity waypoint00 waypoint01)`. First, *stateL* is requested to the Execution Module. Curiosity will retrieve its current *stateL*, indicating that the position of Curiosity is $(x = 0.00, y = 0.00, z = -1.50)$, no bumper detected a collision, and no object is detected to be close by its sonar ring.

Once *stateL* is retrieved, the PELEA flow continues. *stateL* is translated into *stateH* and checked by the Monitoring module. After the monitoring process is done, the Low-level planner is executed in order to generate the next *planL*. The high-level action being executed (in our example : `(navigate curiosity waypoint00 waypoint01)`) is sent to the Low-Level Planner and the corresponding *planL* is generated.

For the case of our `(navigate curiosity waypoint00 waypoint01)` action, the corresponding

[3] Currently, we have already uploaded PELEA with ROS into two humanoid robots, NAOs, and we are using PELEA to control them for simple tasks

```
<plan id="navigate">
   <action-plan name="turnleft">
      <term name="robot" value="curiosity"/>
   </action-plan>
   <action-plan name="movetowardsy">
      <term name="robot" value="curiosity"/>
      <term name="y" value="0.00"/>
   </action-plan>
</plan>
```

Figure 5: *planL* returned

planL for the actuators is shown in Figure 5. Curiosity has to change its orientation turning left. And then it has to move towards the position $y = 1.0$. Now that the *planL* has been generated, the robot can execute the set of commands to achieve the complete level action. First, Curiosity turns left and then moves towards position $y = 1.00$ (PELEA sends a command `movetowardsy` as the one shown in Figure 5). Once the action is executed, the resulting *stateL* is returned by the EM. This state is translated into *stateH* that should look like the fragment on Figure 6.

```
<atom predicate="at">
  <term name="curiosity"/>
  <term name="waypoint01"/>
</atom>
<atom predicate="full">
  <term name="curiositystore"/>
</atom>
<atom predicate="calibrated">
  <term name="logitechsph"/>
  <term name="curiosity"/>
</atom>
...
```

Figure 6: *stateH*

Related Work

In the late 80's and beginning of the 90's there was a profusion of architectures for autonomous mobile robot systems which heavily drew upon the popular three-layer SPA (sensing-planning-acting) architecture. Most of these early approaches (RAP (Firby 1987), PRS (Georgeff and Lansky 1987), SHAPIRA (Saffiotti, Konolige, and Ruspini 1995)) developed reactive behaviors, fixed pre-compiled patterns of actions which are selected depending on the actual situation of the executor. Other architectures smoothly integrate planning and reacting running asynchronously a classical AI planner in conjunction with a reactive control mechanism (Gat 1992; Hayes-Roth 1995). An alternative direction is to generate plans to solve the entire planning task and monitor their execution afterwards, resorting to re-planning or plan repairing in case of execution failures (Currie and Tate 1991). The Continuous Planning and Execution Framework CPEF (Myers 1999) is an asynchronously working architecture that interleaves planning and execution. CPEF can generate plans to arbitrary levels of refinement and then be manipulated at runtime by the executor component. PELEA gathers many of the features of the aforementioned architectures, namely reactive execution, continuous planning approach, re-planning and repairing techniques but it also features a learning module and the ability to change goals as new information is acquired during plan execution (Goal&metric generation module).

PELEA also has some features in common with the Task Control Architecture TCA (Simmons 1992). TCA was developed to handle robot control problems and was specifically tested with Ambler, a robot designed for planetary explorations, which required a rather deliberative architecture. TCA provides a general framework of hierarchical task decomposition augmented with reactive behaviors. Since TCA was specifically thought of as a high-level robot operating system, it allows for the definition of task-specific modules. This contrasts with the more general PELEA behavior, which has been designed to tackle any type of planning-execution problem, from the most proactive to the most reactive domains. PELEA also uses PDDL and HTN definitions that are currently the standards in deliberative planning, so it can more easily be used by planning practitioners.

IDEA (Intelligent Distributed Execution Architecture) (Aschwanden et al. 2006) is a real-time architecture that departs from the three-layer SPA architecture and proposes instead to unify deliberation and execution under a single planning technology and model representation. Like T-Rex, IDEA is composed of self-contained planning systems, each with a deliberation latency and planning horizon. IDEA uses XIDDL, a XML encoding of IDEA domain definition language, but it does not allow for PDDL or HTN-based model representation. Additionally, this unified view that permits the planner to be embedded within the executor usually allows only for a strict and controlled interleaving of the planning and execution phases (Vidal and Nareyek 2011), making it difficult to have a general-purpose planner for different types of executor systems.

As a whole, PELEA boosts flexibility, modularity, generality and interoperability. PELEA allows practitioners to replace (and reuse) any module of the architecture as long as the new module is able to read the corresponding XML input file, thus requiring much less effort to easily generate new interleaved planning-and-execution applications.

Conclusions

In this paper, we have introduced a domain-independent architecture, PELEA, that integrates planning related processes, such as sensing, planning, execution, monitoring, replanning and learning. We have shown examples of the two levels of reasoning in a temporal and no-temporal domains. High level as in a regular automated planning task; and low level, composed by atomics actions that are executed directly in the environment. PELEA is conceived as a flexible and modular architecture that can accommodate state of the art techniques that are currently used in the overall process of planning. This kind of architectures will be a key resource to build new planning applications, where knowledge engineers will define some of the components, parameterize others, and reuse most of the available ones. This will allow engineers to easily and rapidly develop applications that incorporate planning capabilities. We believe this kind of architecture fills part of the technological gap between planning techniques and applications.

References

Ai-Chang, M.; Bresina, J.; Charest, L.; Chase, A.; Hsu, J.-J.; Jonsson, A.; Kanefsky, B.; Morris, P.; Rajan, K.; Yglesias, J.; Chafin, B.; Dias, W.; and Maldague, P. 2004. MAPGEN: Mixed-initiative planning and scheduling for the Mars Exploration Rover mission. *IEEE Intelligent Systems* 19(1):8–12.

Alcázar, V.; Guzmán, C.; Prior, D.; Borrajo, D.; Castillo, L.; and Onaindia, E. 2010. PELEA: Planning, learning and execution architecture. In *PlanSIG'10*, 17–24.

Aschwanden, P.; Baskaran, V.; Bernardini, S.; C. Fry, M. M.; Muscettola, N.; Plaunt, C.; Rijsman, D.; and Tompkins, P. 2006. Model-unified planning and execution for distributed autonomous system control. In *Workshop on Spacecraft Autonomy: Using AI to Expand Human Space Exploration*. AAAI Press.

Blockeel, H., and Raedt, L. D. 1998. Top-down induction of first-order logical decision trees. *Artificial Intelligence* 101(1-2):285–297.

Cesta, A.; Cortellessa, G.; Fratini, S.; and Oddi, A. 2009. Developing an End-to-End Planning Application from a Timeline Representation Framework. In *IAAI-09. Proceedings of the 21st Innovative Applications of Artificial Intelligence Conference, Pasadena, CA, USA*.

Chien, S. A.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *AIPS*, 300–307.

Coles, A.; Fox, M.; Halsey, K.; Long, D.; and Smith, A. 2009. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence Journal* 173(1):1–44.

Currie, K., and Tate, A. 1991. O-Plan: the open planning architecture. *Artificial Intelligence* 52(1):49–86.

De la Rosa, T.; Jiménez, S.; Fuentetaja, R.; and Borrajo, D. 2011. Scaling up heuristic planning with relational decision trees. *Journal of Artificial Intelligence Research* 40:767–813.

Eyerich, P.; Mattmller, R.; and Rger, G. 2009. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proc. ICAPS 2009*.

Fdez-Olivares, J.; Castillo, L.; García-Pérez, O.; and Palao, F. 2006. Bringing users and planning technology together. experiences in SIADEX. In *Proc. ICAPS 2006*. Awarded as the Best Application Paper of this edition.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3:251–288.

Firby, R. J. 1987. An investigation into reactive planning in complex domains. In *AAAI*, 202–206.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In *Proc. ICAPS 2006*, 212–221.

Fritz, C., and McIlraith, S. A. 2007. Monitoring plan optimality during execution. In *ICAPS*, 144–151.

García-Olaya, A.; de la Rosa, T.; and Borrajo, D. 2011. Using relaxed plan heuristic to select goals in oversubscription planning problems. In *Advances in Artificial Intelligence*.

Garrido, A.; Guzman, C.; and Onaindia, E. 2010. Anytime plan-adaptation for continuous planning. In *PlanSIG'10*, 62–69.

Gat, E. 1992. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *AAAI*, 809–815.

Georgeff, M. P., and Lansky, A. L. 1987. Reactive reasoning and planning. In *Proceedings of AAAI-87 Sixth National Conference on Artificial Intelligence*, 677–682.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research* 20:239–290.

Gerkey, B.; Vaughan, R.; and Howard, A. 2003. The player/stage project: Tools for multi-robot and distributed sensor systems. In *ICAR 2003*.

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *2nd International Conference on AI Planning Systems*.

Haslum, P. 2006. *Admissible Heuristics for Automated Planning*. Ph.D. Dissertation, Linkopings Universitet.

Hayes-Roth, B. 1995. An architecture for adaptive intelligent systems. *Artif. Intell.* 72(1-2):329–365.

Hsu, C.-W.; Wah, B. W.; Huang, R.; and Chen, Y. 2007. Constraint partitioning for solving planning problems with trajectory constraints and goal preferences. In *IJCAI'07*.

McGann, C.; Py, F.; Rajan, K.; and Olaya, A. G. 2009. Integrated planning and execution for robotic exploration. In *Procs. of International Workshop on Hybrid Control of Autonomous Systems*.

Myers, K. L. 1999. Cpef: A continuous planning and execution framework. *AI Magazine* 20(4):63–69.

Quintero, E.; Alcázar, V.; Borrajo, D.; Fdez-Olivares, J.; Fernndez, F.; Ángel García-Olaya; Guzmán, C.; Onaindia, E.; and Prior, D. 2011. Autonomous mobile robot control and learning with pelea architecture. In *PAMR'11*, 51–56. AAAI Press.

Saffiotti, A.; Konolige, K.; and Ruspini, E. H. 1995. A multivalued logic approach to integrating planning and control. *Artif. Intell.* 76(1-2):481–526.

Simmons, R. 1992. Concurrent planning and execution for autonomous robots. In *IEEE International Conference on Robotics and Automation*, 46–50.

Vidal, E. C. J. E., and Nareyek, A. 2011. A real-time concurrent planning and execution framework for automated story planning for games. In *Intelligent Narrative Technologies*.

Younes, H. L. S., and Littman, M. L. 2004. PPDDL1.0: An extension to pddl for expressing planning domains with probabilistic effects. Technical Report CMU-CS-04-167.