PELEA: Planning, Learning and Execution Architecture

Vidal Alcázar
Universidad Carlos III de Madrid
valcazar@inf.uc3m.es

César Guzmán
Universidad Politécnica de Valencia
cguzman@dsic.upv.es

David Prior
Universidad de Granada
dprior@decsai.ugr.es

Daniel Borrajo
Universidad Carlos III de Madrid
dborrajo@ia.uc3m.es

Luis Castillo
Universidad de Granada, Granada
L.Castillo@decsai.ugr.es

Eva Onaindía
Universidad Politécnica de Valencia
onaindia@dsic.upv.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 that include sensing, planning, executing, monitoring, replanning and even learning from past experiences. In this paper we describe the design of such an architecture, PELEA (Planning, Execution and LEarning Architecture) that has been conceived as a general-purpose architecture suitable for a wide range of problems from robotics to emergency management. We present the requirements of this architecture, its main components, as well as the connections among them. Currently, we have a first prototype of such platform.

Introduction

Planning technology is being used for many different kinds of applications, ranging from space (Ai-Chang et al. 2004), fire extinction (Fdez-Olivares et al. 2006), logistics (Florez et al. 2010), or education (Castillo et al. 2009) among many others. The process of developing the final application is an "ad-hoc" manual process that requires expertise and techniques from several fields (planning, controllers, learning, or user interfaces among others), as well as the careful definition of the underlying architecture. Most applied work defines architectures that conceptually incorporate a set of common abilities and are structured in a similar way. More specifically, applications are based on sensing the state (which is commonly used in robotics applications, but is also common at different levels of abstraction for all applications), generating the problem at hand, planning (using many different kinds of techniques), executing the plan (by either setting up tasks to a machine, or suggesting actions to a human), monitoring the execution for failures (unexpected results, unability to execute the next action or plan, ...). 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.

Currently, there are some initial attempts to generate generic architectures that have been used for different purposes, as it is the case of 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 particular 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).

The goal of the PELEA project is to build a component-based architecture able to perform planning, execution, monitoring and learning in an integrated way, in the context of PDDL-based and HTN-based planning and suitable for a wide range of planning problems. We define first the architecture, its component modules, as well as the connections among those modules. The architecture would allow the planning engineers to easily generate new applications that integrate all such capabilities by reusing and modifying the components. A second scientific advantage of such architecture would be to allow researchers or practitioners to compare techniques. We intend to provide a set of tools that implement different techniques for each module, so that users can choose among those. The paper describes the on-going work on this architecture.

Overview of PELEA Architecture

The architecture for PELEA includes components that allow the applications to dynamically integrate planning, execution, monitoring, replanning and learning techniques. In general, there are 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. Thanks to this, PELEA can be used to implement applications in the whole spectrum:

- In full deliberative applications (as for instance in the case of applications with no need to respond in *short*

real time), there is no need for a reactive component, so the related components can be set to null. Examples are logistics applications that plan a sequence of trucks movements (Florez et al. 2010).

- In some deliberative applications, there is no need for a reactive component, but it is useful to separate high-level reasoning of some low-level implementation of that reasoning. For instance, in some robotics applications, there might not be a need for the robot to react fast, but it might be useful to separate the specification of high-level actions (navigate, take-image) from their current implementation or multiple implementations (using low-level actions to set the speed of wheels). In this case, it is useful to have these two reasoning levels separated in the two components of the architecture (high-level and low-level) that are usually implemented using different techniques (PDDL-based planning vs. controllers based on all kinds of technologies).
- In full reactive applications, the deliberative component might not be needed or can be used very rarely to set up general plans to carry out. In that case, most of the control loop will be in charge of the low-level components.

It would be possible to add additional levels to allow developers for a more hierarchical decision process. However, we consider that the sole distinction between high and low level is enough to tackle most problems, as has been shown in many robotics applications. Figure 1 shows the current version of the architecture that permits the integration of the modules. Even if we did not provide the explicit APIs, all modules in the architecture have access to either the high-level and low-level domain. We will describe in more detail later on which domain is input and output of each component.

As we can see, it 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 KI because of its wide spread use as a common language to exchange information and our previous experience using it in different real world applications (Fdez-Olivares et al. 2006; Florez et al. 2010; Castillo et al. 2009).

The main KIs that we have used are (the modules also exchange the information related to the parameters that configure how each module works[1]):

- stateL: low-level state composed of the sensory information
- stateH: high-level state, that gets translated from stateL as an aggregation or a generalization of low level information
- goals: the set of high-level goals to be achieved by the architecture
- metrics: the metrics that will be used in the high-level planning process
- planH: set of high level plans. Each high level plan is a set of ordered actions resulting from the high-level planning process. Usually, they will be sequentially ordered, though parallel plans can also be given. The actions of these plans 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)
- planL: set of low level plans. Each low level plan is again an ordered set of actions resulting from the low-level planning process. Usually, it will consist of only one plan, and several actions that can be executed in parallel. These actions should be operational: directly executable in the environment
- domainH: definition of actions for high-level planning
- domainL: definition of behaviors (skills) for low-level planning
- learning examples: to be used by the learning component to acquire knowledge for future planning episodes, either in the form of heuristics, domain models, or knowledge on the problem specification
- heuristics: in different forms (control rules, policies, cases, macro-actions, etc.) allow the planners to improve their efficiency in solving future planning episodes
- monitoringInfo: meta knowledge on the plan that helps to perform the monitoring (as, for instance, the footprint of each action)

[1]For instance, which planner to execute.

Control Flow and Communication

PELEA follows a continuous planning approach, i.e. an ongoing and dynamic process in which planning and execution are interleaved (Myers 1999; Chien et al. 2000). The general algorithm of PELEA is depicted in Algorithm 1, and we detail the flow of the architecture next.

The PELEA architecture is controlled by a module, called Top-level control, which coordinates the execution and interaction of the Execution and Monitoring modules. PELEA architecture uses a two-level knowledge approach. 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). For example, in the blocksworld domain, the operation (stack A B) is a high-level knowledge item, and specifically defines a high-level action.

However, since high-level knowledge descriptions are rarely directly executable, if ever, they must be complemented by the low-level knowledge, which specifies how the operations are actually performed in terms of

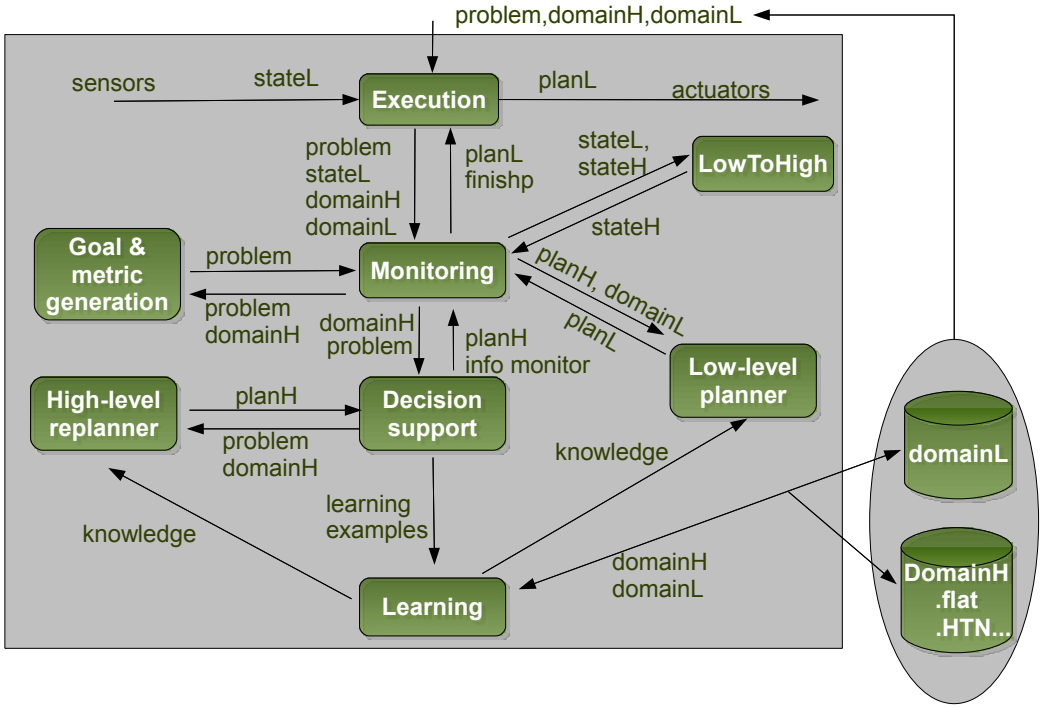

Figure 1: Architecture of PELEA.

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. The low-level knowledge is read from the environment through the sensors placed in the Execution module. The environment is either a hardware device, a software application, a software simulator, or a user. An example of low-level knowledge would be "the coordinates of a robot" or "degrees of motion of a robot arm". In PELEA, it is not necessary to work at the two knowledge levels. For instance, 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 needed.

The starting point of the architecture is the Execution module, which is initialized by the Top-level control (see function *Start* in Algorithm 1), receiving a high-level and low-level domain, and a problem, composed of an initial state, a set of goals to achieve, a set of objects, and, optionally, a metric. The Execution is initialized with the domain and the problem, which in turn initializes the objects and their positions in the environment. The Execution keeps only the static part of the initial state, given that the dynamic part, called *stateL* (low-level state), will come from the environment through the sensors (this is done through the function *getSensors* in Algorithm 1). *stateL*, the problem and the domain are sent by the Top-level control to the Monitoring module to obtain a low-level plan (*planL*) (function *getPlanL* in Algorithm 1). The actions in *planL* are executed one by one by the Execution module (function *ExecuteAction* in Algorithm 1). As commented above, the modules LowToHigh and Low-level planner are only used in case the domain is modeled at the high and low levels. Otherwise, the Monitoring calls directly the Decision Support to obtain a high-level plan (*planH*). On the other hand, the module Goals&Metric Generation is invoked in case the problem goals or the metric change dynamically along the plan execution.

Once the Monitoring module receives the necessary knowledge (state, problem and domain), it starts the monitoring process. The first step of the plan monitoring is to check whether the problem goals have already been achieved (*goalsL* and *goalsH* in case we are dealing with the two processes). If so, the plan execution finishes; otherwise, the Monitor begins with the first iteration of the plan monitoring.

At the first iteration of the algorithm, there is no plan to monitor yet, so the Monitoring calls the Decision Support, which obtains a valid plan that achieves the goals from the current observed state through the High-level replanner. This latter module receives a problem and a high-level domain (*domainH*), and generates a

Data: problem=(stateH, goalsH, metrics, objects), domainH, &optional domainL
Result: finish execution
begin
;; Initializes the Execution, which in turn initializes the simulator
$Execution.Start(problem, domain)$
$planL \longleftarrow null$
repeat
;; Receive sensors from environment (Execution)
$stateL \longleftarrow Execution.getSensors()$
;; If a new plan is returned, it means that the objectives have not been achieved and the new plan must be executed. Otherwise it may be assumed that the¡ZA objectives have been achieved.
$planL \longleftarrow Monitoring.getPlanL(stateL, problem, domainH, domainL, planL)$
;; Execute the actions one by one or the comprehensive plan
if *exists actionL in planL to be executed* **then**
$Execution.ExecuteAction(actionL)$
end
until *¬not (exists actionL in planL to be executed)*
;; When we get here that means that goalsL are satisfied
$finishexecution$
end

Algorithm 1: Top-level control algorithm of PELEA.

high-level plan (*planH*). *planH* is sent back to the Decision Support module, which computes the variables to be monitored and keeps this information in the parameter *info monitor* (Figure 1). Both *planH* and *info monitor* are sent by the Decision Support to the Monitoring.

The Monitoring module, with the help of the Low-level planner module, generates a set of executable low-level actions (*planL*), if this is the case. If the Low-level planner module is not being used, the Monitoring assumes that the high-level actions in *planH* are executable, and they are sent to the Execution module, which executes the actions one by one. Then, it senses the dynamic part of the state from the environment. The Monitoring receives the information from the observed state (*stateL*) after the execution of an action, and verifies the information in *stateL* against the parameter *info monitor*. If the values of all the checked variables are within the value range specified in *info monitor*, the Monitoring continues with the plan execution. Otherwise, if a discrepancy between the expected and the observed state (*stateL*) is encountered, the anomaly is reported to the Decision Support, which determines whether the discrepancy is relevant to the plan execution or not. That is, whether the plan is still valid to achieve the goals from the current observed state. At this point, the low-level planner can also be invoked to find the most immediate actions for a rapid intervention -if reactivity is needed- since this module typically stores predefined behaviours or courses of actions for reaching a situation. In case the Decision Support finds the anomaly entails a plan failure, and so the plan is no longer executable, it will take a decision about whether applying a plan repair, or replanning through the High-level replanner, thus starting a new iteration of the algorithm. Particularly, the Decision Support decides whether it is worth repairing the plan, in which case it fixes *planH* and makes it executable again, or, it would be better to replan, in which case it requests a new plan to the High-level replanner module. In case that the discrepancy is not relevant to the plan validity, the Decision Support resumes the execution of *planH* by sending back the remaining and the new parameter *info monitor* to the Monitoring module, which in turn sends the next action to the Execution.

Whilst no discrepancies are found in the observed state, the two modules that are continuously interacting are the Monitoring and the Execution. The Monitoring not only checks for discrepancies but also if the problem goals (*goalsL* and *goalsH*) are already satisfied in the current state. In that case, the overall process is finished.

XML Schema definition

The PELEA architecture uses a PDDL-like syntax language. However, having components that dynamically integrate planning, execution, monitoring, replanning and learning techniques, we opted for a better option in the form of the Extensible Markup Language (XML). Using XML as the communication standard format has several advantages; eg. XML is considered as a standard reference language, it enjoys vast third-party library support, it allows for a PDDL standard representation, and it is easily extensible and flexible. Our XML Schema definition is called XPDDL, and it is represented by three XML document schemas: domain, problem and plan schemas. The domain schema handles the formal definition of a domain, which is defined

by our XML schema XPDDL. The domain sections are: requirements, types, predicates, and action-def. An example can be seen in Figure 2.

The transformation of the PDDL domain format to XML (XPDDL) is performed by a module component (PDDL2XML) of the DS module, which uses the domain scheme for the translation into this format. The transformation of XPDDL to PDDL is performed by a module component (XML2PDDL) of the High-Level Replanner (HLR).

The problem schema specifies a problem instance of that domain, which describes the objects, init and goal sections. Again, the translation from/to XPDDL to PDDL is performed by the PDDL2XML and XML2PDDL components.

The plan schema encodes the solution to a problem. It also contains meta-data such as the domain and problem names, the number of actions, the initial time, the planning time, and the description of each of the actions. In this case, the transformation of the ASCII plan format to XPDDL is performed by PLAN2XML.

Components of PELEA

In this section we describe in more detail the components that are currently operative in PELEA, namely the Execution Module, Monitoring, Decision Support and High-Level Planner. We also provide some hints on the future incorporation of the Learning module into PELEA.

Execution Module

The Execution Module (EM) is in charge of the interaction between PELEA and the environment. The environment can be either a software simulator, a hardware device (robot), a software application, or a user. In particular, the EM acts as a wrapper over anything external to PELEA, solving issues like communication, data protocols, etc... The tasks of the EM are:

- to initiate PELEA by receiving a particular domain and problem to be solved;
- to observe the current world information (which is composed of the sensors readings) and send it as the low-level state; and
- to send the low-level actions to the actuators

The main algorithm followed by the EM can be seen in Algorithm 1 for the top level control algorithm. The EM communicates with the Monitoring Module (MM) by sending the low-level state to the MM when asked, and receiving actions to be executed from the MM. Usually, these two steps are interleaved, becoming the main execution loop. Currently, the EM provides integration with the following environments:

- MDPSim: PPDDL (Younes and Littman 2004) simulator used in the probabilistic track of the International Planning Competition (IPC). The simulator generates states stochastically based on the probabilistic version of domains. After receiving an action from PELEA's EM, it sends a new state back to PELEA through the EM. Additional support to add uncertainty to deterministic domains has been implemented, too.
- Virtual Robot Simulator (VRS[2]): freeware software suite for robotics applications, research and education. VRS is able to simulate industrial robots manipulators, mobile robots, walking robots, etc.
- Microsoft Robotics Studio and Player: robot independent platforms for controlling robots of various kinds.
- Alive: open platform for developing social and emotion oriented applications (Fernández et al. 2008).
- TIMI (Florez et al. 2010): planning tool for real logistic problems.
- ORTS:[3] a domain independent game platform that is used as a testbed for game development.

Monitoring Module

The task of the MM is to check whether the observed state coming from the Execution is a correct state according to the planning process. The starting point for the plan monitoring is to know which aspects of the plan need to be monitored during execution. Since it is neither possible nor efficient to monitor all the state variables, we have to select the deciding variables both in the context of the plan monitoring and the environment. This information is provided by the Decision Support module in the *info monitor* parameter whose contents are: i) the variables to be monitored, which can be of two types, those directly related to the plan and those related to the environment, ii) the value range for each variable, denoting the set of correct values the variables can take on, and iii) the instants of time at which the variables should be monitored.

The MM also acts as a plan dispatcher sending the EM the actions to execute. At time t there may be n actions to be executed in parallel in the plan; in such a case, the MM sends the n actions to the EM and requests the current real state at relevant times at which effects produced by the actions are expected, most typically at the end of the execution of each action. Unlike approaches that work with temporal flexible plans, the occurrence time of the start/end of actions is simulated to be at fixed time schedules in our model, as returned by PDDL-based planners.

We use a goal regression approach for monitoring the execution of plans and reacting to execution failure, similar to PLANEX strategy in (Fikes, Hart, and Nilsson 1972). The core idea is to represent plans in a way that supports monitoring by representing the plan structure in a triangle table and executing the strategy,

[2]http://robotica.isa.upv.es/virtualrobot/

[3]http://skatgame.net/mburo/orts/index.html

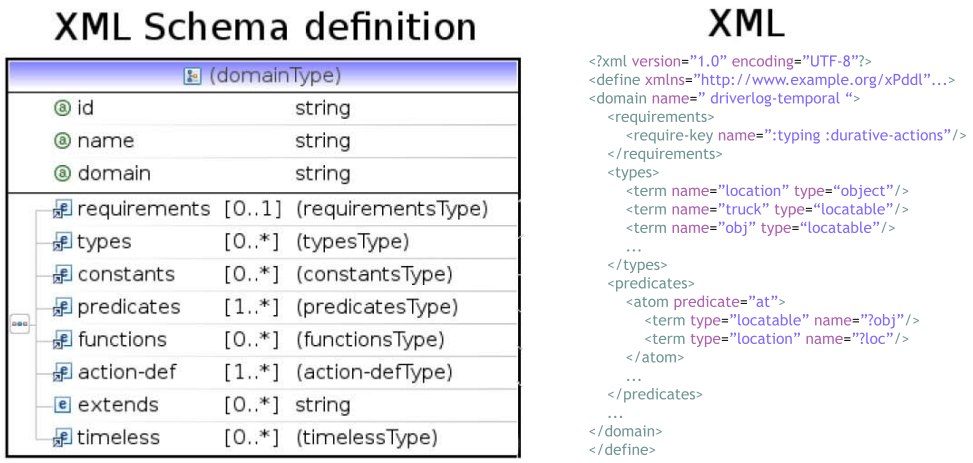

Figure 2: Domain XML Schema definition and XML domain DriverLog.

which consists on regressing the problem goals (including action preconditions) through the remaining operators of the plan. Roughly, the regression of a formula over an action is a sufficient and necessary condition for the satisfaction of the formula following the execution of the action (Fritz and McIlraith 2007).

We have implemented an extension of the goal regression method proposed in (Fikes, Hart, and Nilsson 1972) to deal with the kind of variables that come up in a PDDL2.1 problem specification. Thus, unlike PLANEX, our method is not limited to monitoring sequential plans but parallel or temporal plans. For this purpose, for each variable v to monitor, we record four values in the parameter *info monitor*: a) the time at which v is generated, b) the earliest time at which v is expected to be used as an action precondition, c) the latest time at which v is expected to be used, and d) the value range for v. This way, we can record and monitor any type of temporal variables.

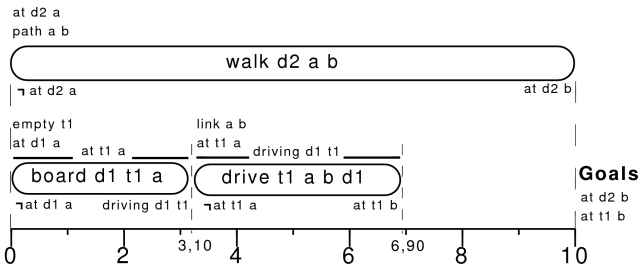

Figure 3: Application example.

An example of the interaction between the EM and MM can be observed in Figure 3. At time *0* the MM sends the two actions (*walk d2 a b*) and (*board d1 t1 a*) to the EM. At time *3,10* the MM requests the current real state and checks against the *info monitor* variable whether the effects of the action (*board d1 t1 a*) have been successfully generated. If there is no discrepancy, the MM sends the next action (*drive t1 a b d1*) to execute, and makes the next requests at times *6.90* and *10* to check the effects of the corresponding actions. Note the EM does not perform a real-time simulation, but it is a discrete event simulator, where events are the start/end of the execution of actions.

Decision Support

The objective of the Decision Support (DS) module is twofold. First, calculate which variables, along with their valid range, need to be monitored by the MM; and, second, activate a deliberative process when the MM reports a discrepancy between the observed state and the expected planning state (including the case in which there is no available plan). Specifically, a discrepancy is found if, according to the information in the parameter *info monitor*, the value of the variable does not fall within the value range at the specified time instant. When a discrepancy is reported from the MM to the DS, the tasks of the DS are:

- check whether the discrepancy is relevant to the plan validity or not; so far, as we are only working with high-level knowledge, every discrepancy found by the MM is considered as a plan failure by the DS.
- if the discrepancy causes the plan not to be longer executable, the DS must make a decision about applying a plan-repair method or replanning from scratch; and
- once a decision is made, and regardless of the choice, the DS will use the High-level Replanner. If the choice is replanning, the DS will invoke the planner with a new problem consisting of the new initial state (observed state) and the problem goals. If the choice is repairing, the DS applies a plan-repair method that will eventually call the High-level Replanner.

Plan-repair based on the analysis of plan dependency structures involves identifying the actions that are no longer executable as a consequence of the plan failure. Obviously, in continuous planning, the interest is not in *repairing* the whole plan but fixing the *earliest portion of the plan* as this will be the first to be executed. Likewise, minimizing the number of changes in the original plan, i.e. maintaining plan stability (Fox et al. 2006), is particularly relevant in the earliest portion of the plan as well. This is because when a plan is being executed, the executive has likely committed the earliest part of the plan in terms of equipment, resources or time, and so it is specially willing to respect the commitments induced by the partial execution of the published plan (Cushing and Kambhampati 2005).

The decision between repairing or replanning is done via the application of goal-state heuristic. This heuristic proceeds regressively and generates all the goal states in the plan. Each goal state is actually representing a possible reachable state from which to reuse the rest of the original plan. Thus, the first goal state is the one from which to reuse the totality of the original plan; the subsequent goal states represent reachable states from which to reuse smaller and smaller parts of the original plan. Besides deciding which part of the original plan to reuse, the heuristic computes an approximate plan from the observed state to the goal state and returns an estimation of the composition of the approximate plan with the reused portion of the original plan. In order to decide between repairing and replanning, the algorithm computes the solution with the *first goal state* (**repairing** by keeping the whole original plan), and the solution with the *last goal state* (**replanning** by discarding the whole original plan). The best value is returned as the adopted choice.

High-Level Replanner

This component is intended to find plans for solving a given problem and behaves in different ways. When called with an initial state, a goal, a domain and an empty plan, the module will find a solution plan from scratch and, therefore, behaves as any state of the art planner. This module might also be called with an existing plan partially executed and a temporal mark, just letting know which part of the plan has already been executed. This would also allow the planner to detect the deviation of the perceived state with respect to the expected state. In this case, this module behaves as a replanner. We plan to carry out experiments with several planners, implementing different paradigms (hierarhical, temporal, cost-based, ...). After reviewing the results on metrics and temporal constraints in the *Temporal Satisfying Track* (IPC-2008), we have selected and successfuly used: LPG-TD (Gerevini, Saetti, and Serina 2003), SGPLAN (Hsu et al. 2007), CRIKEY (Coles et al. 2009) and TFD (Eyerich, Mattmüller, and Röger 2009), and we have included them in the PELEA architecture. The High-Level Replanner (HLP) module communicates with the rest of the PELEA architecture through the DS module (see Figure 1).

The objective of HLR is threefold. First, HLR translates input parameteres *problem* and *domainH* from XPDDL to the PDDL3.0. format. The second goal of the HLR is to call one of the planners in the system (LPG-TD, SGPLAN, CRIKEY or TFD). The selected planner will take *problem* and *domainH* as input parameters (already in PDDL format). After execution, planner will return *planH*. The last goal of HLP is to translate *planH* to the XPDDL format. That translation would be the one returned to DS module.

Learning Module

In the research group, we have implemented several machine learning techniques, and the goal will be to integrate those within PELEA. This component will generate two kinds of knowledge (domain models and control knowledge) for two kinds of planners (high-level and low-level). More specifically:

- Domain model learning for high-level planners: we will integrate techniques that have been previously defined for acquiring those models as those for STRIPS representations (Yang, Wu, and Jiang 2007), HTN (Hogg, Kuter, and Munoz-Avila 2009), or probabilistic (Jiménez, Fernández, and Borrajo 2008).
- Control knowledge learning for high-level planners: we will reuse the extensive experience of the group on learning control knowledge in various formats (control rules, policies, macro-operators or cases) (de la Rosa et al. 2009).
- Learning for low-level planners: building low-level planners by using learning has been achieved by reinforcement learning (Kaebling, Littman, and Moore 1996). In this case, there is usually no explicit difference between learning the domain and control knowledge.

Conclusions

In this paper, we have presented the on-going work on building an architecture, PELEA, that integrates planning related processes, such as sensing, planning, execution, monitoring, replanning and learning. It is conceived as a flexible and modular architecture that can accomodate 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.

Acknowledgements

This work has been partially supported by the Spanish MICIIN project TIN2008-06701-C03.

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.

Castillo, L.; Morales, L.; González-Ferrer, A.; Fdez-Olivares, J.; Borrajo, D.; and Onaindía, E. 2009. Automatic generation of temporal planning domains for e-learning problems. *Journal of Scheduling*. ISSN: 1094-6136 (print version), ISSN: 1099-1425 (electronic version), DOI: 10.1007/s10951-009-0140-x.

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.

Cushing, W., and Kambhampati, S. 2005. Replanning: a new perspective. In *Proc. of ICAPS 2005 Poster Program*.

de la Rosa, T.; García-Durán, R.; Jiménez, S.; Fernández, F.; García-Olaya, A.; and Borrajo, D. 2009. Three relational learning approaches for lookahead heuristic search. In *Proceedings of the Workshop on Planning and Learning of ICAPS09*.

Eyerich, P.; Mattmüller, R.; and Röger, 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.

Fernández, S.; Asensio, J.; Jiménez, M.; and Borrajo, D. 2008. A social and emotional model for obtaining believable emergent behavior. In Traverso, P., and Pistore, M., eds., *Artificial Intelligence: Methodology, Systems, and Applications*, volume 5253/2008 of *Lecture Notes in Computer Science*, 395–399. Varna, Bulgaria: Springer Verlag.

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

Florez, J. E.; García, J.; Álvaro Torralba; Linares, C.; Ángel Garcia-Olaya; and Borrajo, D. 2010. Timiplan: An application to solve multimodal transportation problems. In Steve Chien, G. C., and Yorke-Smith, N., eds., *Proceedings of the 2010 Scheduling and Planning Applications woRKshop (SPARK'10)*, 36–42.

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.

Georgeff, M. P., and Lansky, A. L. 1987. Reactive reasoning and planning. In *Proceedings of AAAI-87 Sixth National Conference on Artificial Intelligence*, 677–68.

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.

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

Hogg, C.; Kuter, U.; and Munoz-Avila, H. 2009. Learning hierarchical task networks for nondeterministic planning domains. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09)*. AAAI Press.

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 *Proceedings of IJCAI'07*.

Jiménez, S.; Fernández, F.; and Borrajo, D. 2008. The PELA architecture: integrating planning and learning to improve execution. In *Proceedings of the AAAI'08*. Chicago, IL (USA): AAAI.

Kaebling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *International Journal of Artificial Intelligence Research* 237–285.

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

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence* 171(2-3):107–143.

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, School Of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.