Performance Level Profiles for Code Reuse and Plug'n Play Autonomy

Maor Ashkenazi[1] and Ronen I. Brafman[1]

Abstract— **Autonomous robots, such as service robots, must combine diverse capabilities in diverse ways to perform different tasks. Writing functional modules that supply these capabilities (e.g., navigation, manipulation, object recognition, and much more) requires much expertise and time. The ability to provide and use software packages and to reuse code from different projects is essential for making it possible to build new platforms quickly and reliably. However, the use and integration of software, whether one's own or imported, requires much manual work, including reading documentation, and later on, writing scripts that combine functional modules in useful ways. Recently, we introduced Performance Level Profiles (PLPs), a formal, yet intuitive XML-based language for specifying the behavior of functional modules [1]. Unlike existing, text-based specification formats, PLPs are machine readable and more precise, and support software modularity in a number of ways. First, they provide a form of contract – an important software engineering notion crucial for code reuse – for robotics. More importantly, they take us closer to modular construction and reuse of software for autonomous systems. In [1] we describe the automated generation of monitoring code given PLPs. Here we explain how we support automated reuse and integration of multiple functional modules via AI task-planning technology: Given the PLP specification of a given functional module, we automatically generate wrappers that enable a generic task planner to interact with these modules without modifying their code. The planner can schedule module execution, in essence, providing "scripts" on demand.**

I. INTRODUCTION

Modular design and the ability to (re)use third-party code are essential for building large complex robotic systems. This was quite evident in a recent project for building an autonomous compact track loader (CTL) for discovery and evacuation of land-mines. The project involved a large industrial partner (Israel Aerospace Industries), a professional service provider, and academic partners, each providing software packages that tackle different aspects of the problem. For the industrial partner, replacing its own code base and methodology was not an option, and it complained about the difficulty of marketing products without the ability to provide clear performance guarantees for the CTL's functional modules. With different modules supplied by different partners, documented using the widely used text-based SSS (systems/subsystem specification), it was difficult to foresee how each module would behave. And often, different parties made different assumptions and had different expectations from the software packages.

Beyond these basic integration issues, other problems arose. Various performance problems encountered at run-time were identified at a late stage, although with appropriate monitoring earlier warnings could have been issued. The basic capabilities of navigation, localization, etc. were combined using a man-written script, and each time a mine is detected, an operator must supply an explicit sequence of operations to perform. Moreover, although its basic capabilities can be combined in diverse ways (e.g., clean an area of waste) doing so with its current architecture would require writing a new script.

To address this issue, we suggest a methodology that is based on two components: a formal, yet intuitive XML-based language for describing the behavior of functional modules, together with tools that exploit the machine-readability of this description to generate code automatically. Although it is difficult to provide a precise specification of the behaviour of a functional module given an open-ended world, it appears possible to provide a "good-enough" partial description of the module's role, the conditions under which it is expected to succeed, its success probability, and its expected running time (when relevant). To this end, we define *performance-level profiles* or *PLPs* [1]. Technically, PLPs are XML files, and hence machine readable and parsable. They have a formal syntax, specified by an XML Schema Description file, and are pretty intuitive. One can view them as a more precise and constrained version of free-text specification and documentation formats, adapted for robotic applications.

PLPs describe a number of key aspects of the performance of functional modules. They combine ideas from planning languages [2], [3] with diverse goal notions, such as achievement and maintenance goals [4], and new notions such as progress measures and a *repeat* construct aimed at making explicit the frequency by which input parameters are read and output parameters are published. They can also be viewed as extending the expressivity of contracts within the Design by Contract approach [5] to address the needs of robotics applications.

In this paper, we describe our effort to support plug-n-play planning, facilitating per-task automated integration of existing and new modules, making code reuse simpler – in fact, automated – and making modular design easier. Given a module's PLP, our system does two things: First, it automatically generates wrappers for each module – constructing a middle layer between the users modules and ROSPlan – a ROS-based task-planning tool [6]. This middleware provides an abstraction layer that allows for seamless integration of the modules with ROSPlan, without requiring the user to understand ROSPlan or write code for it. Second, we generate the required input for ROSPlan's planner. This input informs the planner about each module's needs and abilities, and it is used to generate the task plans. Here, some effort

[1]Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel {maorash,brafman}@cs.bgu.ac.il

is required, as the current underlying planner is a classical planner, whereas robots operate under uncertainty and with partial information of the world. This is achieved using domain compilation techniques.

II. PLPs

The primary objective of a PLP is to clarify the role and expected behavior of a module. As a simple example, imagine a module designed to grasp an object. The expected outcome is that the object is held by the arm. However, in most realistic settings, this effect is not guaranteed. There is some probability of failure, and failure can come with some side effects, such as the object falling, or being broken. And while the running time is most likely not deterministic, we can try to describe properties of its distribution. We can also describe the action's rate of progress. For instance, the grasping module is usually not static until the object is captured, and so we expect its position to change at some minimal rate. Moreover, the probability of success and failure may depend on various properties, such as the shape and size of the object. Furthermore, in certain settings it may be unrealistic to specify the success probability, as too many external things could impact it. For example, if another arm is attempting to catch the object at the same time, it may be difficult to predict which one will succeed.

PLPs can be thought of as a rich version of action description languages. They have two abstract components. The first circumscribes the conditions under which the profile is provided – conditions that must hold for it to be valid. And the second specifies the effect of the action. The first part contains a list of input/output module parameters and the frequency in which they are read or written, a list of required resources with constraints on their rate of change, preconditions, and concurrency conditions (conditions on the world during module execution time, including which modules can or cannot execute concurrently with the current module). The second part specifies the meaning of successful execution, describes the possible failure modes and their probability, the distribution over running times, and the expected rate of progress. More generally, it can specify a statistical profile of various aspects of normal module behavior at run-time.

PLPs are described in XML-based format. Each XML document must conform to the schema defined for the corresponding PLP type: *achieve, maintain, observe,* and *detect*. These schema are described using XML Schema Definition (XSD) files. XML and XSD were chosen for their simplicity and wide-spread use and support. Any tool for editing XML and verifying its structure based on an XSD file can be used as a PLP editor. The precise syntax of the four schema is laborious, and can be found in https://github.com/PLPbgu/PLP-repo together with an example of a PLP of each type.

Each PLP type corresponds to different types of functional modules. *Achieve* modules attempt to reach a state of the world in which some desired property holds. For example, fuel tank is full, robot is standing, plane has landed, etc. Achieve also covers cases where the goal is to generate some virtual object, such as a map or a path. Beyond the common elements, their PLP contains the achievement goal – a Boolean condition defined over suitable parameters, failure modes – which are various ways the module could fail to achieve the goal, the probabilities associated with each failure mode, the success probability, and the running time distribution given success and given failure.

Maintain modules attempt to maintain the value of a variable or the truth value of some more complex condition. For example, maintain heading, maintain speed, maintain perimeter clean, etc. The condition need not be initially true, and so the module may need to initially attain the condition, very much like a closed-loop controller that always attempts to decrease some distance to the desired goal condition. Beyond the common elements, their PLP contains: the condition to be maintained, whether it is initially true or not, termination conditions, one for successful termination and one for failure, failure modes, the probability of successful termination and different failure modes, and the runtime distribution given success and failure. A success termination condition is not necessary, and the run-time distribution will often be memoryless (i.e., exponential).

Observe modules attempt to identify the value of some variable(s) or a Boolean condition in the current world state. For example: observe distance to wall, observe whether robot is standing, observe whether object is held. Beyond the common elements, their PLP contains: the observation goal – a Boolean condition to be verified or a parameter whose value is to be observed. We also describe the probability of failure to observe, the probability the observation is correct (if Boolean) or some form of error specification, such as confidence interval and confidence level, and the running-time distribution given success and given failure.

Detect modules attempt to identify some condition that is either not true now, or that is not immediately observable. For example, detect intruder, detect temperature change, detect motion, detect obstacle, etc. Beyond the common elements, their PLP contains: the detection goal – the condition being detected, and the probability the condition will be detected given that it holds (*true* positive) and given that it does not hold (*false* positive).

All four module types also specify side-effects, that is, possible results of executing this module that are not a measure of its success or failure. For example, if the module consumes some resource, a natural side effect is that the level of this resource is reduced. Side effects are described by a conditional assignment to a parameter, which could depend on a local variable (such as running time, or distance traveled). Intuitively, side-effects are changes caused by the module that could potentially impact other modules.

An important performance aspect described by PLPs is *Expected Progress Rate.* Some modules perform continuous work to achieve or maintain their goals. For example, when navigating a robot, its position will change at some rate as long as the robot is not at its destination. Another example, when grasping an object, the arm will move closer to the object. In this field, one specifies the minimal rate of change per time unit, as well as the time unit itself (e.g., $\Delta(x) \geq 1$ meter every 1 minute).

Many robotic modules operate by repeatedly updating or modifying some data-structure or signal based on information that is constantly being updated. To capture this type of behavior, there is also a *repeat* wrapper for the *achieve* and *observe* modules. For example, a path-planning module may update the path as it obtains updated maps. It can be viewed as performing an *achieve* task repeatedly.

For more information about PLPs and their use in performance monitoring, see [1] and the links above.

III. ROSPlan

ROSPlan [6] is a framework that provides a method for task planning in ROS. It combines both planning and plan dispatch. ROSPlan uses an underlying planner in order to generate plans. The current planner is POPF [7], which uses PDDL 2.1 [2]. The framework supports changing a planner easily. ROSPlan is composed of three main parts:

- *Planning System* - handles planning (using the underlying planner), dispatching actions for execution and replanning.
- *Knowledge Management System (KMS)* - holds the PDDL data and data about the state of the world and provides services to manage them.
- *Components* - the action nodes that will be dispatched by the Planning System.

The Components need to subscribe to a shared ROSPlan topic, used for action dispatch messages, and listen to the messages sent from the Planning System. When a message is received, the Components check whether or not they are responsible for the actions being dispatched. If so, they are responsible for updating the Planning System on success or failure of the action. In addition, the Components need to update the KMS on the current world state. According to the above, in order to integrate an already implemented robotic module into ROSPlan, we have two options:

- Modify the module, treating it as a component, adding to it code that interacts with the Planning System and the KMS.
- Leave the module unchanged, and implement a new component that interacts with the Planning System, the KMS and the module, thus creating an abstraction layer between them.

We take the second approach, and exploit the module's PLP description to *automatically* generate the new component.

IV. Using PLPs for Planning

Our basic premise is that the code writer or code user have specified a PLP for the relevant modules. Given this information, we generate the middleware required to allow existing code to communicate with a planner and database – provided by ROSPlan – and the input required by the planner. The compilation method used to generate planner input is somewhat technical, and requires some familiarity with task planning. Thus, due to space constraints, we refer the interested reader to [8], where it is described in detail.

ROSPlan's Components (action nodes) must interact with the planner and KMS, which, as noted earlier, requires the user's understanding of ROSPlan and its interfaces and of PDDL. This goes against the ideal of plug-n-play capability. Instead, we exploit the information in the PLP to automatically generate module dispatchers (*PLP dispatchers,* for short) that dispatch plan steps to the appropriate underlying modules. Each PLP dispatcher is a single ROS node that provides a layer of abstraction (middleware) between the Planning System and the KMS, and the relevant PLP's underlying module. Each PLP dispatcher will be responsible for the following:

1) Listening to the ROSPlan topic used for action execution. Once a relevant message is received, the dispatcher activates the PLP's underlying module's trigger (in order to execute it).
2) Constantly monitoring the termination conditions of the PLP's underlying module. Once a termination condition holds, the dispatcher updates the planning system whether the action failed or succeeded.
3) Receiving the PLP module's output parameters and, if needed, saving them in the KMS. This is crucial because we might need to pass them to another PLP as part of a trigger.
4) Updating the KMS with the current state of the world according to the results of the PLP modules execution and the PDDL actions effect.
5) The current planner input might make certain assumptions, so if a plan fails because of a wrong assumption, the dispatcher updates the assumptions in the KMS and informs the planning system that the action failed, thus triggering a replan.

An interesting extension we are currently implementing is detecting wrong assumptions earlier. This can be achieved by monitoring the results of observation actions to identify inconsistencies between the assumptions and the observed real world values. If an inconsistency between the assumptions and real values is detected, the assumptions will be changed and a plan validator will be used to ensure the plan is still valid. If not, a replan will occur.

Code generation requires, in addition to the PLPs, a *glue* file which maps PLP parameters to PDDL parameters. The PDDL parameters are sent to the PLP dispatchers when an action is dispatched by the Planning System. To trigger a relevant PLP module, the PLP dispatcher needs to send correct execution parameters to the module. The mapping allows us to do this easily. In addition, the KMS is checked to see if any relevant parameter values (for the module's trigger) were stored there from a previous module's execution. For example, an objects location was sent as output from an observe module and is needed for a module that grabs the object.

This middleware allows us to continue adding new code (modules) to our system without changing the previous modules or writing new code, offering users the desired plug-n-play behavior.

V. USE CASE

We tested our ability to integrate a planner into an existing project by utilising it within a project for developing an autonomous service robot, at this stage, working in simulation only. Our robot, Robotican's Armadilo robot, is a mobile platform with a six DOF arm. It was supplied with a number of basic capabilities for object recognition and grasping. Additional development work was done by students, often on top of existing ROS packages. What makes this project particularly suitable to demonstrate the effectiveness of our tools and ideas is the fact that all of the basic abilities were implemented by different developers that know nothing about ROSPlan. In fact, beyond existing ROS packages, the code was written by undergraduates unfamiliar with planning and ROSPlan, who simply implemented new ROS nodes.

In the version of the service robot we worked on, the robot's goal is to bring a cup of coffee from anywhere in its world (as defined by its acquired maps) to a goal location. The robot can perform simple tasks such as: observing a cup of coffee, grabbing a cup of coffee, navigating, observing a button, pressing a button and so on. We wanted to use these basic abilities in a plug-and-play manner in task planning, without changing or implementing new code.

The architecture is shown in Figure 1. First, a PLP was written for each of the basic abilities (modules). Then, the PLP2PDDL compiler was used to generate a PDDL domain file. Next, the code generator generated the middleware PLP dispatchers that are in charge of handling the communication between ROSPlan and the modules built by the developers – their code was incorporated as is. Finally, we supplied ROSPlan with the generated PDDL domain file and the relevant problem file and the Planning System started planning and dispatching the plan. The PLP dispatchers received all of the messages sent by the Planning System, activated the relevant modules and monitored their execution. When a module finished running, the PLP dispatcher informed the Planning System on success or failure and updated the KMS on the current world state. Thus, we demonstrated that given a set of PLPs for some robotic modules, they can be used in a plug-and-play manner in task planning and execution. A screenshot from the project is shown in Figure 2. In addition, a video demo of the project execution is available at: https://github.com/PLPbgu/PLP-repo.

VI. SUMMARY

We described our progress towards plug-n-play integration of task planning into robotics. Plug-n-play of hardware devices in personal computing require the ability of the operating system and device to communicate using a well known protocol, and use it to obtain the needed information. Similarly, for task planning in robotics, we propose the use of *Performance Level Profiles*, a machine-readable description-format for robotic modules. PLPs are motivated by standard planning languages, but attempt to make the specification more intuitive and appropriate by resorting to the well known notions of achievement and maintenance goals, adding also analogous notions for the sensing-side of robot activity. Given the PLPs of a module, our software is able to support its integration as a basic capability available to a planner, as well as the generation of a PDDL description of its capability, which is used as input to the planner.

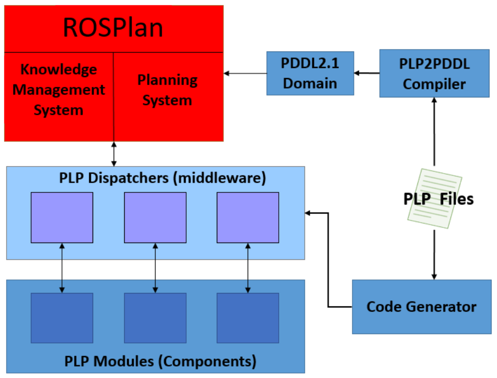
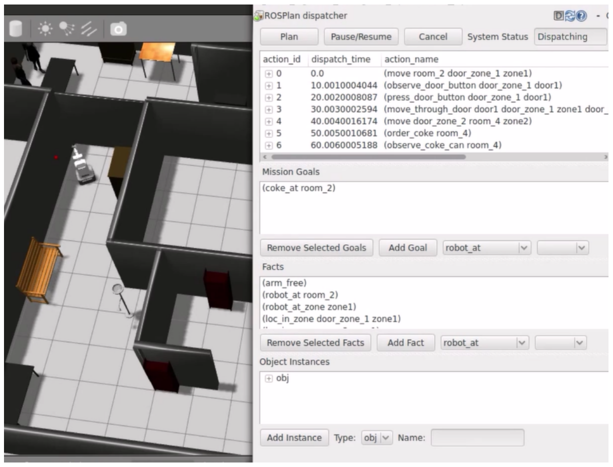


Fig. 1. System architecture



action_id	dispatch_time	action_name
0	0.0	(move room_2 door_zone_1 zone1)
1	10.0010004044	(observe_door_button door_zone_1 door1)
2	20.0020008087	(press_door_button door_zone_1 door1)
3	30.0030002594	(move_through_door door1 door_zone_1 zone1 door_
4	40.0040016174	(move door_zone_2 room_4 zone2)
5	50.0050010681	(order_coke room_4)
6	60.0060005188	(observe_coke_can room_4)

(coke_at room_2)

(arm_free)
(robot_at room_2)
(robot_at_zone zone1)
(loc_in_zone door_zone_1 zone1)

obj

Fig. 2. Using the generated PDDL domain and the PLP dispatcher middleware for ROSPlan to plan and execute a service robot

REFERENCES

[1] R. I. Brafman, M. Bar-Sinai, and M. Ashkenazi, "Performance level profiles: A formal language for describing the expected performance of functional modules," in *Proceedings of International Conference on Intelligent Robots and Systems*, 2016.
[2] M. Fox and D. Long, "Pddl2.1: An extension to pddl for expressing temporal planning domains," *JAIR*, vol. 20, pp. 61–124, 2003.
[3] S. Sanner, "Relational dynamic influence diagram language (rddl): Language description," 2010.
[4] F. Ingrand, R. Catilla, R. Alami, and F. Robert, "A high level supervision and control language for autonomous mobile robots," in *IEEE ICRA*, 43-49, Ed., 1996.
[5] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.
[6] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtós, and M. Carreras, "Rosplan: Planning in the robot operating system," in *Proc. 25th Inter. Conf. on Automated Planning and Scheduling, ICAPS*, 2015, pp. 333–341.
[7] A. J. Coles, A. I. Coles, M. Fox, and D. Long, "Forward-chaining partial-order planning," in *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10)*, 2010.
[8] M. Ashkenazi, M. Bar-Sinai, and R. I. Brafman, "Planning and monitoring with performance level profiles," in *ICAPS'16 Workshop on Planning and Robotics (PlanRob)*, 2016.