Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)

Verifying Plans and Scripts for Robotics Tasks Using Performance Level Profiles

Alexander Kovalchuk, Shashank Shekhar and **Ronen I. Brafman**
The Department of Computer Science
Ben-Gurion University of the Negev, Beer-Sheva, Israel
{kovalchu, shekhar, brafman}@cs.bgu.ac.il

Abstract

Performance-Level Profiles (PLPs) were introduced as a type of action representation language suitable for capturing the behavior of functional code for robotics. This paper considers two issues that PLPs raise: (1) Their formal semantics. (2) How to verify a script or plans that schedule the use of components that have been documented by PLPs. We discuss formal semantics for PLPs that maps them to probabilistic timed automata (PTAs). We also show how, given a script that refers to components specified using PLPs, we derive a PTA specification of the entire system. Using a model checker, we can now verify various properties of the system and answers queries about its behavior. Finally, we empirically evaluate an implemented system based on these ideas that use the UPPAAL-SMC model checker and demonstrate its scalability. The result is a pragmatic approach for verifying various properties of component-based robotic systems.

Introduction

Most robotic systems are built by assembling software components, locally written, or imported, each of which handles a particular capability. A more sophisticated behavior is then obtained by combining these behaviors in various ways. Unfortunately, as noted in Abdellatif et al. (2012): "Systems built by assembling together, independently developed and delivered components often exhibit pathological behavior. Part of the problem is that developers of these systems do not have **a precise way of expressing the behavior of components**..." Addressing this issue is crucial to our ability to deploy autonomous robots in open environments.

Brafman, Bar-Sinai, and Ashkenazi (2016) advocated the use of an intuitive *machine readable* descriptive (rather than normative, or prescriptive) behavior specifications. Such specifications make more precise what must be said and how, and they enable the development of tools that can utilize them to *automatically* support monitoring, validation, and planning. To that effect, they introduced *Performance Level Profiles* (PLPs), a language for specifying the expected behavior of functional components. PLPs describe a number of key aspects of the performance of functional modules. They combine ideas from planning languages (PDDL 2.1 (Fox and Long 2003), probabilistic PDDL (Younes and Littman 2004), RDDL (Sanner 2010)), achievement and maintenance goals (Ingrand et al. 1996; Kaminka et al. 2007), and new notions such as progress measures and a *repeat* construct that makes explicit the frequency by which input and output parameters are read and published. Unlike action languages that limit their expressiveness to meet the requirements imposed by state-of-the-art planning technology, PLPs seek to provide expressiveness that can be used for other tasks. Thanks to their structured, machine readable syntax, PLPs can be manipulated automatically for the purpose of online monitoring (Brafman, Bar-Sinai, and Ashkenazi 2016), validation, and planning (Ashkenazi, Bar-Sinai, and Brafman 2016). In this paper, we describe their use in support of verifying component-based systems.

Code for complex tasks uses complex control structures to schedule code fragments that implement diverse behaviors. Verifying and understanding the properties of the resulting system is crucial if we are to address our original concerns. This paper describes an approach for performing such validation when these code fragments have been documented using PLPs. First, we provide formal semantics for PLPs by mapping them to *probabilistic timed automata* (PTAs) (Beauquier 2003), a model much used in program verification. The full mapping is quite technical, and can be found in (Kovalchuk 2018). Here we present a short introduction to it. Our second step is to describe a rich language for specifying complex scripts, which we refer to as *control graphs*, and a mapping that takes as input a control graph and the PLPs of the components it uses, and outputs a large PTA. We leverage the well known UPPAAL-SMC probabilistic model-checker (David et al. 2015) to verify this PTA, and hence, the original script. We empirically demonstrate the scalability of this approach by experimenting with a software system (freely available) that implements these ideas.

Background

We briefly describe PLPs and PTA. See: Brafman, Bar-Sinai, and Ashkenazi (2016); Beauquier (2003) for more details.

PLPs

The objective of a PLP is to clarify the role and expected/normal behavior of a module. The four PLP types correspond to four module types: *Achieve* modules attempt to achieve a new state of the world or generate a new object.

A simple example is code for changing the orientation of the robot to some goal orientation. *Maintain* modules attempt to maintain some property. A simple example is code for ensuring that the robot remains within some confined area. *Observe* modules attempt to recognize some property of the current state of the world, such as the robot's location, or whether there is a cup on the table. *Detect* modules monitor the state of the world until some condition holds.

Each PLP document must conform to an XML Schema Definition (XSD) that defines its syntax, with one XSD for every PLP type. The schema and an example of a PLP of each type be found in https://github.com/PLPbgu/PLP-repo. Below we provide an informal description of the information contained in the respective XML/XSD documents. We expect programmers or users to provide this documentation but realize that they are unlikely to precisely capture quantitative aspects such as success probabilities. Given recent advancements in reinforcement learning, we expect that a more realistic approach will start with some initial specification by the programmer that is then improved automatically using learning algorithms.

PLPs have two abstract components. One component specifies the code's expected behavior – its "guarantees": what success means, possible failure modes and their probabilities, a distribution over running times, progress rates, and various statistical invariants. The other component provides the conditions under which the "guarantees" are valid: properties of the world before and during execution and constraints on available resources. These properties are not necessarily observable by the robot. For example, a sensor may guarantee a normal operation under some temperature range, independent of whether the robot has a thermostat.

Common Elements All modules specify the following elements: *Parameters* (values supplied to the module as input or provided by the module as its output), *local variables and their ranges*, and the following set of conditions specifying the contexts in which the PLP is valid: required resources, optional bounds on the maximal rate of change for resources, concurrency conditions that must hold at execution time, invariants, other code modules that must or must-not be executed concurrently, and the frequency by which each parameter must be read or written (optional).

Each module has an intended effect or role. However, it may also have side-effects that are a result of executing this module but are not a measure of its success or failure. Resource consumption is a primary example. In addition, modules that perform continuous work to achieve or maintain their goals may specify a minimal rate of change per time unit. For example, the rate of change of a position while navigating. Making these expectations explicit makes it easier to recognize problematic behaviour while the module executes.

PLP Types *Achieve* modules attempt to make some desirable property true or generate some virtual object. For example, fuel tank is full, robot is standing, generate a map, compute a path. Beyond the common elements, *Achieve* PLPs contain an *the achievement goal*, *failure modes*, *probabilities* associated with success and each failure mode, and the *running-time distribution* given success and given failure.

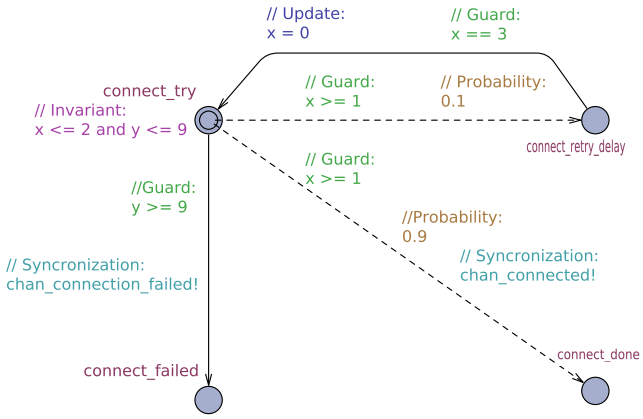

Figure 1: Example PTA for connection protocol

Maintain modules attempt to maintain the value of a variable or property. e.g., maintain speed or maintain perimeter clean. The condition need not be true initially, and so the module may need to initially attain the condition. It may also become false during execution and regained, as in the case of a cleaning robot. This is reminiscent of a closed-loop controller that always attempts to decrease some distance to the desired goal condition. The *Maintain* PLP contains: the *condition* to be maintained, whether it is *initially true*, *termination conditions*, one for successful termination (optional) and one for failure, failure modes, the *probability* of successful termination and different failure modes, and the *runtime distribution* given success and failure.

Observe modules attempt to identify the value of some variable(s), e.g., distance to wall or whether an object is being held. *Observe* PLPs contain additional fields for the *observation goal*, the *probability* of failure to observe, the *probability* the observation is correct or some form of error specification, such as confidence interval, 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 or detect temperature change. Their PLPs contain additional fields for the condition being detected, and the *probability* the condition will be detected given that it holds and given that it does not hold.

PTAs

Probabilistic timed automata (PTAs) (Beauquier 2003) model systems with probabilistic and real-time characteristics. They can be viewed as a combining Markov Decision Processes (MDPs) with a timed-automata. A PTA contains: 1. Integer-valued variables. 2. Clocks – non-negative real-valued variables, which all increase at the same rate. 3. Constraints – boolean combinations of (in)equalities consisting of sums of clock variables and constants. 4. Locations (the PTA's *nodes*) – a finite set of locations, with a distinguished initial location. 5. Actions (the PTA's *edges*) – a finite set of transitions between locations. 6. Invariant conditions – constraints on locations. 7. Enabling conditions – constraints on actions. 8. Probabilities – transition probabilities of enabled

actions. The automaton state consists of the current node and the values of its clocks and variables.

The transition function allows two types of transitions between states: 1. Time transition – all clocks advance by a certain time interval, while the invariant of the current node is preserved. 2. Action transition – transition on an enabled edge chosen according to the probability. As part of the transition, the values of variables and clocks can be updated, too. The automaton starts at the initial node and advances along edges according to invariants and enabling conditions. The state of the PTA describes the current location and the clock values. And a run is legal if each next state is reached by a legal transition: either a change in clock value that is permitted by the invariants, or an action transition in which an edge between locations whose guards are satisfied is traversed and clocks are possibly updated in line with the constraints. The edge probability can be used to associate the likelihood of the transition/run given the action/s.

We use a stronger PTA variant supported by UPPAAL. 1. We support urgent nodes – nodes without time transition such that clocks cannot advance while in them. 2. In basic PTAs, one can only reset clock values to 0. We allow variable values to be updated to any function of other variable values, as well as to a value obtained by sampling some distribution. 3. We use multiple concurrent PTAs. This serves as syntactic sugar, as they can be encoded as a single product PTA. 4. We allow channels. Channels are used to synchronize the transitions of different PTAs. A channel is tied to an edge and can be used either to send or receive a signal. The action of passing a signal on a channel is immediate. Transition on an edge with the sending end of the channel does not delay the transition on that edge, but transition on an edge with a receiving end of a channel may delay the transition until the signal on the channel is received. In addition, we use real-valued variables for convenience, but model them with finite precision as fractions.

Figure 1 describes in graphical form a PTA for a connection protocol with up to three retries. There are two clocks: x and y. Node *"connect_try"* is the initial node, with invariant: $x \leq 2 \wedge y \leq 9$. At anytime in the interval $\left[0,1\right)$, it is impossible to transition along edges because of the guards (enabling conditions). Up until two time units, the PTA can stay at *"connect_try"*, but then it must transition on an edge to *"connect_done"* (with probability 0.9) or *"connect_retry_delay"* (with probability 0.1). If it transitioned to *"connect_done"*, it sends a signal on *"chan_connect"* channel and remains at *"connect_done"* state. If it transitioned to *"connect_retry_delay"*, it waits for x to become 3, then updates x to 0, and transition to *"connect_try"* for another connection attempt. If all three attempts to connect fail, the automaton will transition from *"connect_try"* to *"connect_failed"*, send a signal on *"chan_connection_failed"* channel, and will remain in *"connect_failed"* state.

Related Work

Our semantics for PLPs is obtained by mapping them to a PTA. This type of semantics is sometimes called translation semantics and semantic anchoring. PTAs were used for this purpose in a number of earlier systems: mapping AADL to UPPAAL (Johnsen et al. 2012) and mapping RT-DEVS to UPPAAL (Furfaro and Nigro 2008). Neither of these systems uses the probabilistic aspects of PTAs, and both are part of systems that strive to provide a complete bottom-up approach to robot software design. Our use of PLPs attempts to address systems that use existing, imported, or locally developed components. More recent work (Foughali, Ingrand, and Seceleanu 2019) translates the code written using the Genom3 platform (Mallet et al. 2010) to timed automata. Like the above systems, Genom3 is a complete platform for designing robot software. Unlike the above, and similar to our work, this work also introduces the option of using a probabilistic model of the environment, by essentially learning the frequency of different outcomes within an originally non-deterministic model. They then use UPPAAL-SMC to do probabilistic model checking on the resulting PTA. Finally, probably closest to our work is (Lesire, Doose, and Grand 2020). It describes a language for describing skills that are quite similar to PLPs, although it does not have probabilistic components. From this description, they can generate PDDL descriptions and use planning to compose skills, much like (Ashkenazi, Bar-Sinai, and Brafman 2016) as well as finite-state machines, which are then used for verification using the NuSMV model-checker (Cimatti et al. 2002).

The composition of simple components to obtain more complex ones is a basic technique in automata theory, supported by operations such as Cartesian product and automata sequencing (Hopcroft, Motwani, and Ullman 2003). Tree structures specifically, are often used to describe hierarchical compositions and branching computation. Our work uses these ideas, but supports general graphs with loops.

The idea of verifying systems viewed as trees or graphs of processes or components is not new in robotics. In Simmons, Pecheur, and Srinivasan (2000) the authors develop an approach for verifying elements of the Task Description Language (TDL) (Simmons and Apfelbaum 1998) related to decomposition and synchronisation. This is done by providing a translation into the SMV model-checking language. In Armbrust et al. (2013) behavior networks are verified by using model-checking. In Heinsemann and Lange (2018), TSL, a domain specific language for robotics, which makes use of task trees and hierarchical decomposition, like TDL, is verified by translating its specifications into a Promela model used by the Spin model checker. More recently, ASPiC (Lesire and Pommereau 2018) is a system that allows the composition of simple petri-nets to obtain complex control structures/plans. Combining the ability to verify petri-nets with the semantics of the composition operators used, the system is able to verify a certain form of soundness. The basic elements scheduled by these languages differ significantly from PLPs. First, none of these methods model stochastic elements, while PLPs make use of probabilistic information, and control graphs allow for probabilistic choice, modeling stochastic environments and, consequently, require the use of probabilistic model checking. Second, PLPs offer more information about run-time behavior (e.g., progress measure, run-time distributions), are divided into four categories based on the component's role, yet

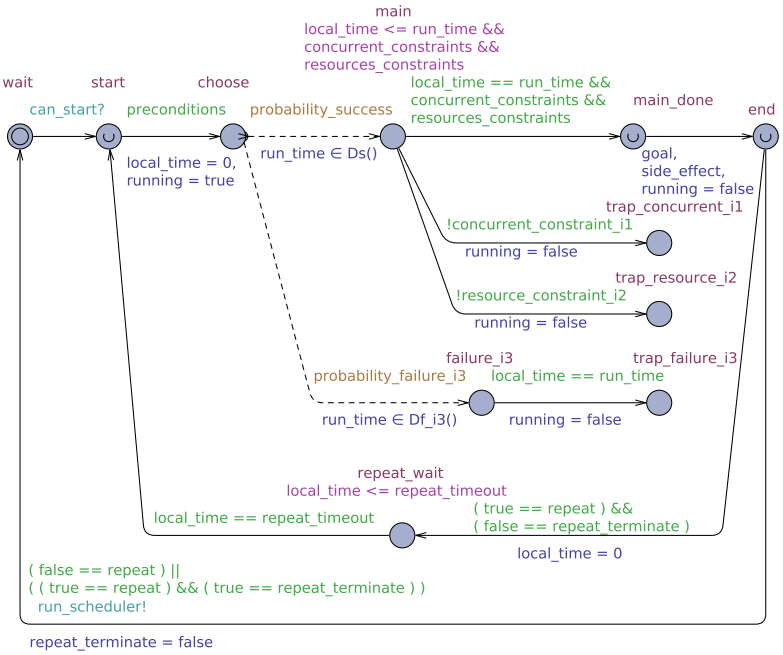

Figure 2: Template for PTA_{PLP} achieve

are not rich enough to actually allow for code generation, as in these methods.

Formal Semantics for PLPs

Compared to PLPs, PTAs are a much more detailed, program-like description of behavior. As such, they can be used for code specifications or programming controllers. PLPs, on the other hand, aim to provide a more abstract, intuitive description of implemented code. Given this, it is natural to use PTAs as a semantic model for PLPs. Here, we outline a translation semantics for PLPs by mapping them into PTAs. Due to space limitations, we provide an overview of the semantics of Achieve. The complete specification appears in Kovalchuk (2018).[1]

(1) For every PLP type, a distinctive PTA scheme exists, but all share a common structure that we describe here.

The successful execution path of each PTA contains the following sequence of nodes: 1. *"wait"* – waits for the scheduling PTA to let the current PTA command run (i.e., the relevant code modeled by the PLP is starting to execute).[2] 2. *"start"* – the scheduling PTA allowed the current PTA command to run. 3. *"choose"* – the PLP's preconditions hold. 4. *"main"* – run-time path for a successful execution. 5. *"main_done"* – PLP's code terminated. 6. *"end"* – completed current execution cycle of PTA_{PLP}.

Node transitions are as follows: 1. *"wait→start"* is taken when a signal from the scheduling PTA is received. 2. *"start→choose"* is taken only if the preconditions are fulfilled. 3. *"choose→main"* is taken when the PTA selects (probabilistically) to take the success path. 4. *"main→main_done"* is taken when run time is up, and the concurrent and resource related constraints are fulfilled. 5. *"main_done→end"* updates side effects and goal conditions. However, if the probabilistic choice in the *choose* node leads to a failure path, a PTA_{PLP} may end up in one of the PLP's failure states.

(2) For a given set of PLPs representing a certain system, we list all variables, parameters, constants, and resources. Then, we match a PTA_{PLP} variable for each and initialized them based on the initial values specified in the PLPs. We also create one status variable for each PLP. It is used to indicate whether the PTA_{PLP} is currently running or not.

(3) In PLPs the concept of a *condition* is used in two distinctive ways: 1. A logical expression such as $a = b$ that must be satisfied by the external world; for example, a precondition. 2. A logical expression that is made true by the code module modeled by the PLP – which is essentially an assignment, such as in the case of goal conditions.

Logical conditions are transformed to negation normal form, and are then translated to a PTA guard condition of an appropriate edge in the PTA_{PLP}. Assignments are translated to a PTA update of an edge in the representative PTA_{PLP} according to its role and place in the PLP.[3] Unfortunately, our translation does not support existential and universal quantifications, at present.

For a given *achieve* PLP, we create PTA_{PLP} achieve, described in Figure 2. PTA_{PLP} starts at the initial node, *wait* and

[1] Available at *https://github.com/a-l-e-x-d-s-9/Thesis2017/blob/master/ThesisToLatex/Thesis.pdf*

[2] The scheduling PTA captures the controller that selects when to execute a module. Later, we define an explicit scheduler model. Here, we simply treat it as an external entity that decides when to activate a PLP.

[3] Recall that action transitions perform such updates.

waits for a start signal from the scheduling PTA. When a signal arrives on channel *"can_start"*, it transitions to *"start"*. First, we check the PLP's preconditions by a transition to *"choose"* with guard condition *"precondition"*. Then, the PTA samples a successful execution or one of the possible failure modes based on the associated probabilities. If a failure path is chosen, it waits for *"run_time"* time according to run-time distribution *"Df_i3()"* in *"failure_i3"* node and then stays in *"trap_failure_i3"* state. If a successful execution is chosen, it transitions from *"choose"* node to *"main"*. Time can pass in the *"main"* node according to the run time distribution *"Ds()"* stored in the *"run_time"* variable.

Even if the current path represents a successful internal execution, external constraints may still force the PLP to fail. This is captured by *"main"*'s invariant condition *"concurrent_constraints && resources_constraints"*. In case of failure caused by a concurrent condition, concurrent module, or resource, the PTA transitions to *"trap_concurrent_i1"* or *"trap_resource_i2"* respectively. If the external constraints are fulfilled while in *"main"*, the transition to *"main_done"* is possible. Finally, the transition to *"end"* node updates the goal conditions and side effects.

PLP logical conditions are converted to PTA guard conditions as follows: 1. Preconditions to *"preconditions"*. 2. Concurrency conditions and concurrent modules constraints are transformed into m_1 statements that are conjoined to form *"concurrent_constraints"*. For each statement $i_1 \in [1, m_1]$ there is a path to *"trap_concurrent_i1"* from *"main"* with a guard *"!concurrent_constraint_i1"*. 3. Required resources are gathered into m_2 statements and conjoined to form the *"resources_constraints"*. For each statement $i_2 \in [1, m_2]$, there is a path to *"trap_resource_i2"* from *"main"* with a guard *"!resource_constraint_i2"*. 4. The *Repeat* state of a PLP is represented by a boolean variable *"repeat"*.

Assignment conditions in the PLP are converted to assignments in the PTA as follows: 1. The transition between *"main_done"* to *"end"* updates the PLP's goal condition to true. (This is the "goal" statement there.) 2. The definition of side effects in PLPs does not specify the time in which the side effect occurs, and we therefore decided to make this change immediately before the PTA completes the transition between *"main_done"* and *"end"*.

Constraints between concurrent modules are enforced by using the *"running"* status variable. This is a flag that indicates, for each PTA_{PLP}, whether its underlying module is currently being executed. Every PTA can include guard conditions that refer to the running state of another PTA, and thus either restrict or require its concurrent execution.

Verifying Complex Controllers

Given a controller that calls different code modules, for each of which we have a PLP, we generate a set of interacting PTAs that represent the entire program. These PTAs can be fed into UPPAAL-SMC (David et al. 2015), a PTA verification tool, which can be queried to verify various conditions. Below we describe our formalization of such controllers and the main ideas behind their mapping to PTAs.

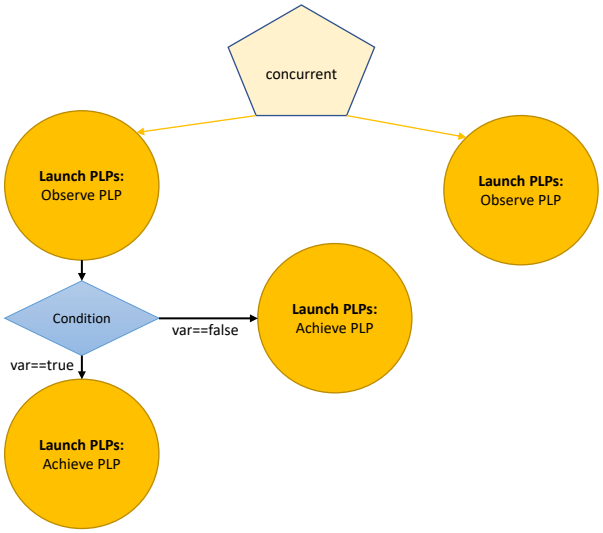

Figure 3: Second Part of First Control Graph

Control Graphs

We use *control graphs* to describe algorithms controlling execution of robotic modules specified by PLPs. A control graph is a directed graph with a single root node in which execution starts. Each node represents a call to some code modules. Transitions between nodes depend on the system state obtained when the parent node(s) terminates execution. They can be stochastic or conditional on the current state.

There are four types of control nodes: 1. PTA_{PLP} launcher nodes launch a sequence of PTA_{PLP} that executes one at a time. 2. Probabilistic nodes choose a single edge to proceed with based on the edge's probability. This allows implementing methods that require some randomization, e.g., to escape from cycles. 3. Conditional nodes choose a single edge to proceed with. Only an edge whose condition is satisfied can be selected. If more than one edge condition is satisfied, one is selected non-deterministically. (If no condition is satisfied, then this is viewed as a failure.) 4. Concurrent nodes execute all nodes reached by their outdoing edges concurrently. Finally, each node type comes in two variations: 1. Starts only when **all** of its immediate parents terminated. 2. Starts whenever **at least one** of its parents terminated. Nodes in the control graph can update the value of variables that are used both by PTA_{PLP} and other nodes in the control graph.

An important aspect of control graphs is the ability to express loops by allowing backward edges. This allows us to specify a much larger class of algorithms. Circular execution can be ended by a probability node or a conditional node.

The Control Graph Verifier

To verify control graphs with PLPs, we produce a set of PTAs representing this system. We then use UPPAAL-SMC to answer queries about the system. UPPAAL is a software package for modeling, validation, and verification of real-time systems modeled as networks of timed automata, extended with data types (Behrmann et al. 2006). UPPAAL-SMC is its extension for stochastic model checking.

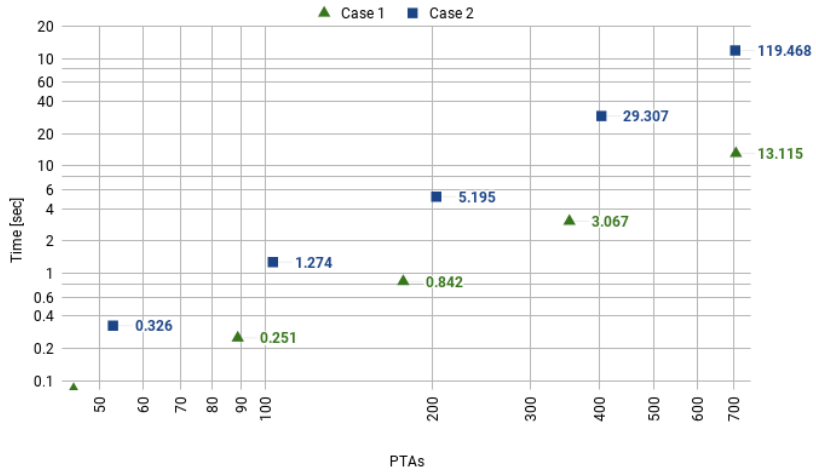

Figure 4: Average Time for ∃ Query

UPPAAL allows us to query temporal properties of the whole system such as: 1. Possible reachability: Is there an execution path in which p will be eventually true? 2. Guaranteed reachability: Will p be eventually true in all execution path? 3. Safety: Will p be true at all times in all execution path? 4. Possible safety: Is there an execution path in which p will always hold? 5. Conditional versions of 1-4. 6. Probability of reachability: What is the probability that p will eventually be true? 7. Probability of an invariant: What is the probability that p will always be true?

Queries of the form 6 and 7 are the most interesting from the robotics perspective. Queries of type 6 allow us to ask what is the probability that we will reach a certain goal state with the given controller. For example, what is the probability that coffee will be served eventually? They also allow us to quantify the probability of a particular risk. For example, what is the probability that we will run out of gas? Queries of type 7 allow us to ask safety queries: what is the probability that some invariant will be maintained throughout execution, e.g., what is the probability that our autonomous car will not take part in an automobile accident?

To convert control graphs to a network of PTAs, we associate a PTA with each node of the graph (PTA_{Node}). $\text{PTAs}_{\text{Node}}$ exist alongside PTAs_{PLP} and can influence each other through shared variables and channels. This (quite technical) mapping appears in (Kovalchuk 2018).

Empirical Evaluation

To evaluate our implemented system's performance and scalability, we run a scalability study and a use-case study.

We evaluate resource demands of the system in two phases: The first phase evaluates the scalability of the compilation to UPPAAL of the PTAs associated with the control graph and its PLPs. The second phase evaluates the scalability of verification queries on the compiled system.

To evaluate both phases we use two independent test cases. The first test case is a comprehensive control graph with most of our functional elements, all types of control nodes and all PLPs types except *detect PLP*. The control graph consists of a full binary tree of probabilistic nodes, such that each of its leaf nodes is associated with the independent control sub-graph shown in Figure 3. By controlling the binary tree's height, we change the graph's size.

The root node of the sub-graph associated with each leaf node allows concurrent execution of two paths: the first path contains a *maintain* PLP that maintains a certain condition needed by the other execution path. The other execution path executes an *observe* PLP, which is followed by a conditional node whose choice depends on the previously observed variable. This conditional node leads to the execution of an *achieve* PLP that achieves a certain goal, but it also requires the concurrent *maintain* PLP to run at the same time.

The second test case is a simple control graph with a single node that launches a sequence of PLPs. All PLPs are functionally identical but recognized by the system as unique. It is an extreme form of a PTAs tree, with all PTAs concentrated along a single path, contrary to the first test case with a full and balanced tree of PTAs. The first test case can be more challenging to compile due to an abundance of elements and connections. The second test can be more challenging for query evaluation.

The results presented below were obtained on a system running an Intel Core i7-4700MQ CPU, 2.40GHz × 8 with 16 GB RAM, SSD, Java 1.8.0 171, and Ubuntu 17.10 64bit. We used "verifyta" – terminal based query verifier of UPPAAL for Linux – 4.1.19. Results are averaged over ten runs.

Generally, every PLP and *control node* in the system are converted to a single PTA in UPPAAL. In the first test case, this number increases exponentially with height. To make the two test cases comparable, in the second test case we use a total number of PTAs similar to the first.

We tested the compilation process with up to 90,000 PTAs, although we cannot envision, in the near future, a system with more than a few dozen components. For 90,000 PTAs, compilation large number of 1000 PTAs compilation required 7.2MB and 4MB of RAM for cases 1 and 2, respectively. Figure 5 describes the run-time of the compila-

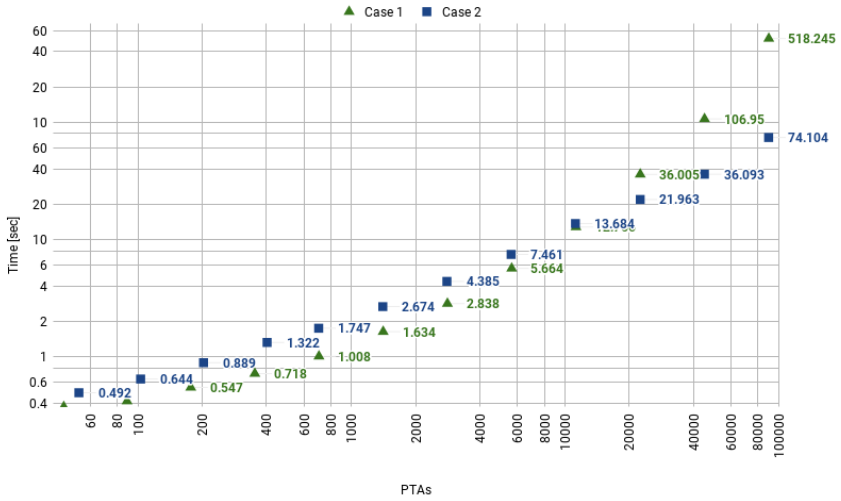

Figure 5: Average Compilation Time

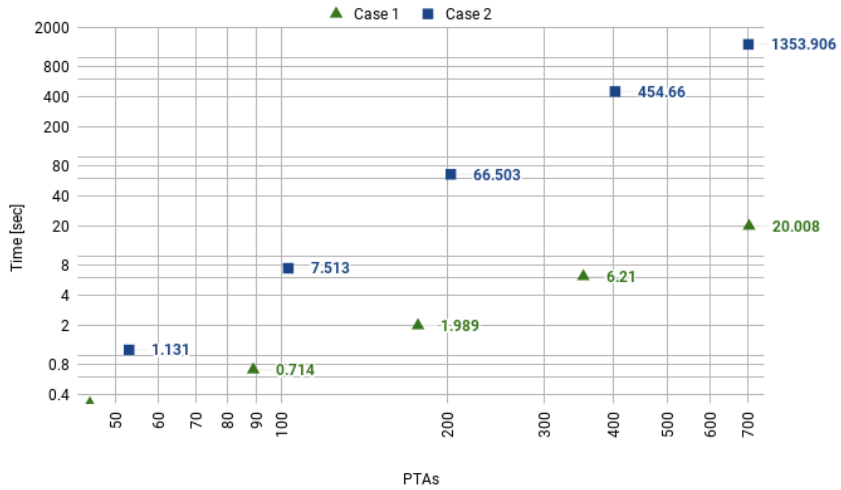

Figure 6: Average Time for Probability Query

tion process. The results clearly indicate that compilation, a one-time process, is quick and scales to very large problems.

Once the system is compiled, we can test its properties using UPPAAL queries. The time and memory needed by UPPAAL depends on the query and on the properties of the PTAs graph affecting the length of paths and number of paths needed to evaluate the query. Therefore, results for specific query types may vary even in the same system.

First, we evaluate a path existence query ("E <>"). For both test cases, we test whether the system can reach the most distant PTA_{PLP} from the initial state. Run time is shown in Figure 4. Query cost does not scale up as well as compilation cost, and systems with over 700 PTAs cause UPPAAL to crash due to stack overflow. Yet, for moderate system sizes, it is relatively efficient, and multiple queries can be carried out in reasonable time. In fact, in systems with ≤ 200 PTAs, online queries for evaluating plans can be supported. Recall that the number of PTAs roughly corresponds to the number of PLPs, i.e., to the number of basic robotic capabilities used in the code.

The second query we tested was a probability ("Pr <>") query of successfully reaching the most distant PTA_{PLP} from the initial state. We defined each PLP with one failure and one success path, both with certain probabilities. UPPAAL calculates probability by multiple evaluations (i.e., by sampling runs), which may take a long time. The results are show in Figure 6. Again, we see that query time for smaller models is reasonable. Certainly, verifying controller properties off-line is realistic, and on, e.g., a service robot operating in the home environment without severe time pressure, online evaluation is possible, too.

Service Robot Case Study. Next, we evaluated our system on a simulated real-world scenario. ARMadillo – a service robot at our lab, is to serve coffee to a person in the audi-

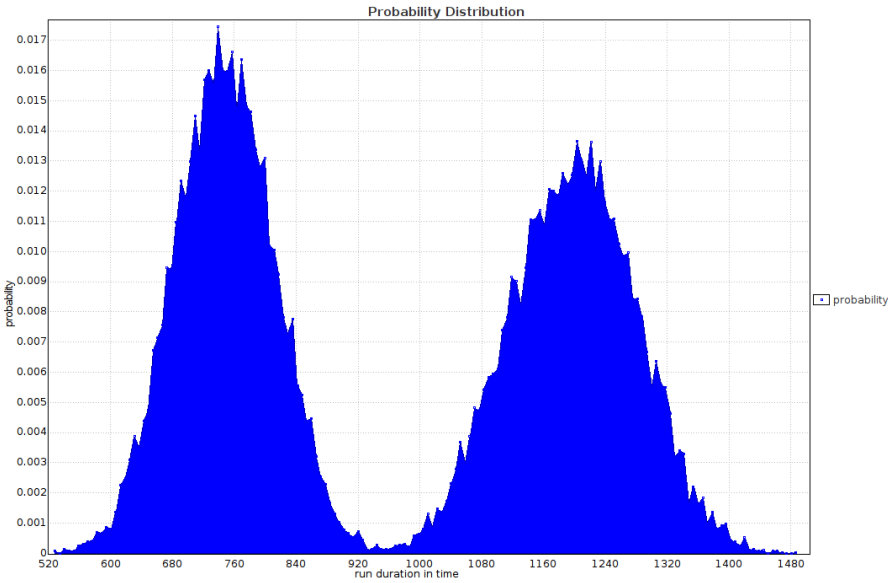

Figure 7: Run time distribution of the system for Query 1.

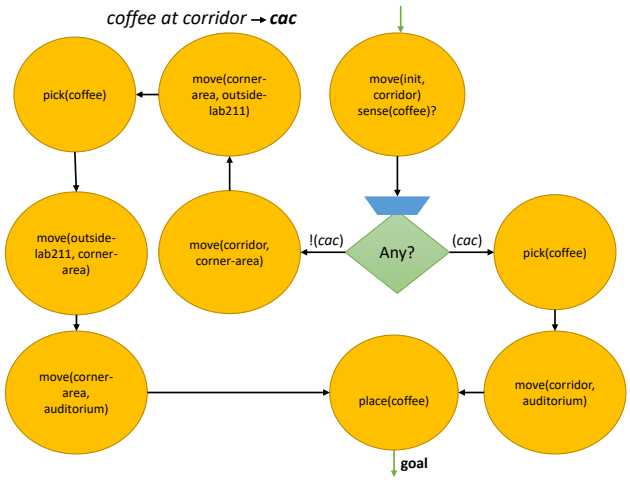

Figure 8: The control graph for the service robot example.

torium. The robot's possible actions are *observe* coffee cup, *navigate* to different locations at our office floor, and *pick* and *place*. The robot starts near the elevator and the coffee is either at corridor area or outside lab211. Its control graph is shown in Figure 8. The robot executes an *achieve* PLP, followed by an *observe* PLP, followed by a *conditional node* that branches into two sequences of *achieve* PLPs.

We compiled the control graph and the PLPs for the demo's code into an UPPAAL input file. We assumed all the actions eventually succeed, except for the *place* action that fails with probability 0.1. Query 1 asks for the probability that the person will get the coffee before it gets cold, where we used 25 minutes as the time limit. The output generated is a probability range [0.895, 0.90] with confidence of 99.5%. Figure 7 describes the PDF over this time steps, showing two peaks – one for each path of the control graph. The query took 15.15 seconds and 56.2 Mb of memory. If we reduce the time limit to 20 minutes, then the probability range drops to [0.67, 0.68] with confidence of 99.5%. This query took 34.36 seconds and 56.7 Mb of memory. Query 2 asked what is the probability that the coffee will ultimately be served? UPPAAL took 11.72 seconds and 47.1 Mb of memory to return the probability range [0.896, 0.905] with a confidence of 95% within 100 thousands time units (i.e., $\Pr[\leq 100000](<> \ldots)$).

Summary

We gave an overview of formal semantics for *Performance Level Profiles* (PLPs) that maps PLPs to probabilistic timed automata. Because PLPs have a formal syntax, they are machine-readable and processable, allowing us to leverage these semantics to map actual PLPs to PTA specifications. We extended this mapping to cover compositions of PLPs within complex control structures and used UPPAAL-SMC to answer queries about plan properties. Our empirical evaluation shows the potential of our approach, although online query evaluation is probably too slow without additional optimizations. For example, we can perform verification in parallel with plan execution.

We firmly believe that the vision of associating machine-readable documentation with code, in addition, or instead of standard natural-language documentation is realizable, especially given recent advancements in RL and natural language processing algorithms. This paper demonstrates just one of the advantages of such documentation, which we believe can lead to far greater support for safe autonomous robots, without requiring drastic changes in the practice of writing code.

Acknowledgements

This work was supported by ISF Grants 1651/19 and 1210/18, by the Israel Ministry of Science and Technology Grant 54178, and by the Lynn and William Frankel Center for Computer Science.

References

Abdellatif, T.; Bensalem, S.; Combaz, J.; de Silva, L.; and Ingrand, F. 2012. Rigorous design of robot software. *Robotics and Autonomous Systems* 60(12): 1563–1578.

Armbrust, C.; Kiekbusch, L.; Ropertz, T.; and Berns, K. 2013. Tool-assisted verification of behaviour networks. In *ICRA 2013*, 1813–1820.

Ashkenazi, M.; Bar-Sinai, M.; and Brafman, R. I. 2016. Planning and Monitoring with Performance Level Profiles. In *ICAPS'16 Workshop on Planning and Robotics (PlanRob)*.

Beauquier, D. 2003. On Probabilistic Timed Automata. *Theor. Comput. Sci.* 292(1): 65–84.

Behrmann, G.; David, A.; Larsen, K. G.; Håkansson, J.; Pettersson, P.; Yi, W.; and Hendriks, M. 2006. UPPAAL 4.0. In *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*, 125–126. IEEE Computer Society. doi:10.1109/QEST.2006.59. URL https://doi.org/10.1109/QEST.2006.59.

Brafman, R. I.; Bar-Sinai, M.; and Ashkenazi, M. 2016. Performance Level Profiles: A Formal Language for Describing the Expected Performance of Functional Modules. In *IROS*, 1751–1756.

Cimatti, A.; Clarke, E. M.; Giunchiglia, E.; Giunchiglia, F.; Pistore, M.; Roveri, M.; Sebastiani, R.; and Tacchella, A. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV 2002*, 359–364. Springer.

David, A.; Larsen, K. G.; Legay, A.; Mikucionis, M.; and Poulsen, D. B. 2015. Uppaal SMC tutorial. *Int. J. Softw. Tools Technol. Transf.* 17(4): 397–415. doi:10.1007/s10009-014-0361-y. URL https://doi.org/10.1007/s10009-014-0361-y.

Foughali, M.; Ingrand, F.; and Seceleanu, C. 2019. Statistical Model Checking of Complex Robotic Systems. In *Model Checking Software - 26th International Symposium, SPIN 2019*, volume 11636, 114–134. Springer.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR* 20: 61–124.

Furfaro, A.; and Nigro, L. 2008. Embedded Control Systems Design based on RT-DEVS and temporal analysis using UPPAAL. In *Proceedings of the International Multiconference on Computer Science and Information Technology, IMCSIT 2008*, 601–608.

Heinsemann, C.; and Lange, R. 2018. vTSL – A Formally Verifiable DSL for Specifying Robot Tasks. In *IROS'18*.

Hopcroft, J. E.; Motwani, R.; and Ullman, J. D. 2003. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley. ISBN 978-0-321-21029-6.

Ingrand, F.; Catilla, R.; Alami, R.; and Robert, F. 1996. A High Level Supervision and Control Language for Autonomous Mobile Robots. In 43-49, ed., *IEEE ICRA*.

Johnsen, A.; Lundqvist, K.; Pettersson, P.; and Jaradat, O. 2012. Automated Verification of AADL-Specifications Using UPPAAL. In *14th International IEEE Symposium on High-Assurance Systems Engineering*, 130–138.

Kaminka, G. A.; Yakir, A.; Erusalimchik, D.; and Cohen-Nov, N. 2007. Towards Collaborative Task and Team Maintenance. In *Autonomous Agents and Multi-Agent Systems*.

Kovalchuk, A. 2018. A Formal Semantics for PLPs using Probabilistic Timed Automata, and its Application to Controller Verification using UPPAAL. Ben-Gurion University of the Negev.

Lesire, C.; Doose, D.; and Grand, C. 2020. Formalization of Robot Skills with Descriptive and Operational Models. In *IROS'20*.

Lesire, C.; and Pommereau, F. 2018. ASPiC: An Acting System Based on Skill Petri Net Composition. In *IROS 2018*, 6952–6958. IEEE.

Mallet, A.; Pasteur, C.; Herrb, M.; Lemaignan, S.; and Ingrand, F. 2010. GenoM3: Building middleware-independent robotic components. In *IEEE ICRA*, 4627–4632.

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description.

Simmons, R.; and Apfelbaum, D. 1998. A Task Description Language for Robot Control. In *IEEE IROS*.

Simmons, R. G.; Pecheur, C.; and Srinivasan, G. 2000. Towards automatic verification of autonomous systems. In *IROS 2000*, 1410–1415.

Younes, H.; and Littman, M. 2004. PPDDL1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects. Technical Report CMU-CS-04-167, Carnegie Mellon University.