Verifying Plans and Scripts for Robotics Tasks Using Performance Level Profiles

Alexander Kovalchuk and Ronen I. Brafman
Ben-Gurion University of the Negev, Beer Sheva, Israel
{kovalchu,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 addresses two issues that PLPs raise: (1) Their formal semantics. (2) How to verify a script or a plan that schedule the use of components that have been documented by PLPs. We provide a formal semantics for PLPs by mapping 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. This PTA can be used to verify the system's properties and answers queries about its behavior. Finally, we empirically evaluate an implemented system based on these ideas, demonstrating its scalability. The result is a pragmatic approach for verifying 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. 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.

In Brafman, Bar-Sinai, and Ashkenazi (2016), we advocated for the use of 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, we introduced *Performance Level Profiles* (PLPs) (Brafman, Bar-Sinai, and Ashkenazi, 2016), 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 language (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 aimed at making explicit the frequency by which input parameters are read and output parameters are 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 schedules diverse behaviors using complex control structures – conditionals, parallel execution, randomness, and loops. Verifying and understanding the properties of com-

plex scripts that schedule existing code fragments 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.

Our first step is to provide formal semantics for PLPs by mapping them to *probabilistic timed automata* (PTAs) (Beauquier, 2003), a model that is much used in program verification. We build on this semantics to provide a mapping from scripts whose primitive actions invoke code modules for which a PLP exists, to PTAs. 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. Next, we leverage existing tools for verifying PTAs to verify 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. For additional details, see: Brafman, Bar-Sinai, and Ashkenazi (2016); Ashkenazi (2017); Beauquier (2003).

PLPs

The primary objective of a PLP is to clarify the role and expected/normal behavior of a module. There are four PLP types, corresponding to four module types. *Achieve* modules attempt to achieve a new state of the world or generate a new object. For example, changing the orientation of the robot to some goal orientation. *Maintain* modules attempt to maintain some property. For example, maintaining some orientation; or, ensuring that the robot remains within some confined area. *Observe* modules attempt to recognize some property of the current state of the world. For example, the robot's location, or whether there is a cup on the table. Finally, *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 the syntax of PLPs, with one XSD for every PLP type. The schema can be found in https://github.com/PLPbgu/PLP-repo together with an example of a PLP of each type. Below we provide an informal description of the information contained in the respective XML/XSD documents. While we expect programmers or users to provide this documentation, they are unlikely to be able to provide precise descriptions of quantitative aspects such as success probabilities. Given recent advancements in reinforcement learning, we expect that a more realistic approach will combine some initial specification by the programmer that is then improved automatically using learning algorithms.

PLPs have two abstract components. The second 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 first 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 necessarily observable by the robot. For example, a sensor may guarantee 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 reach a state of the world in which some desirable prop-

erty 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 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.

Maintain modules attempt to maintain the value of a variable or the truth value of a boolean condition, 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. PLP of *maintain* modules 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) or a Boolean state condition, e.g., distance to wall or whether an object is 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 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 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 (*true* positive) and given that it does not hold (*false* positive).

PTAs

Probabilistic timed automata (PTAs) model systems with probabilistic and real-time characteristics (Beauquier, 2003). A PTA resembles a nondeterministic finite automaton reinforced with integer-valued variables, probabilistic transitions, and a concept of time passage. In its basic form, a PTA consists of the following sets: 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 between states allows two types of transitions: 1. Time transition – an advancement of all clocks by a certain time interval, while the invariant of a 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 advance through edges according to invariants and enabling conditions.

We use a stronger variant of the PTA model supported by UPPAAL. 1. Urgent nodes are also allowed. An urgent node is a node without time transition such that clocks cannot advance while in it. 2. In basic PTAs, one can only reset clock values to 0. In our model, variable value can be updated to a value which is a function of other variable values, as well as 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. 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 tran-

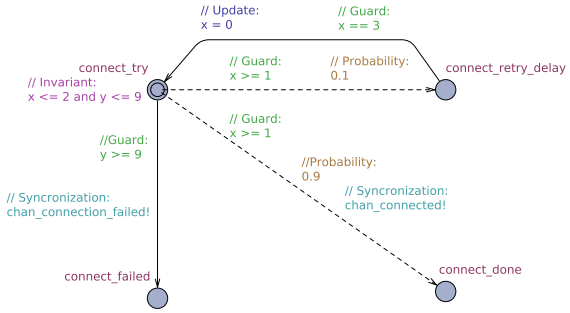

Figure 1: Example PTA for connection protocol

sition 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. In our figures, we use the following color codes:

Attribute	Color
Node name	
Node invariant	
Edge guard	
Edge probability	
Edge update	
Edge synchronization	

Related Work

Our semantics for PLPs is obtained by mapping them to 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 use 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, Fleury, and Bruyninckx, 2002) 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 is 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 soundess. 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 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 specification or for 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 skip over some details and describe only Achieve and Maintain. See Kovalchuk (2018) for the complete specification.[1]

(1) For every PLP type, a distinctive PTA scheme exists, but all schemes 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 successful execution. 5. *"main_done"* – PLP's code terminated. 6. *"end"* – completed current execution cycle of PTA_{PLP}.

The transitions between nodes 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 the side effects and goal conditions.

Besides its successful execution path, a PTA_{PLP} may end up in one of the PLP's failure states. This occurs if the probabilistic choice in the *choose* node leads to one of the failure path.

(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. PTA_{PLP} variables are initialized according to the initial values specified in the PLPs. We also create one status variable for each PLP that is used as an indication of 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.

Modelling Achieve PLP

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

[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.

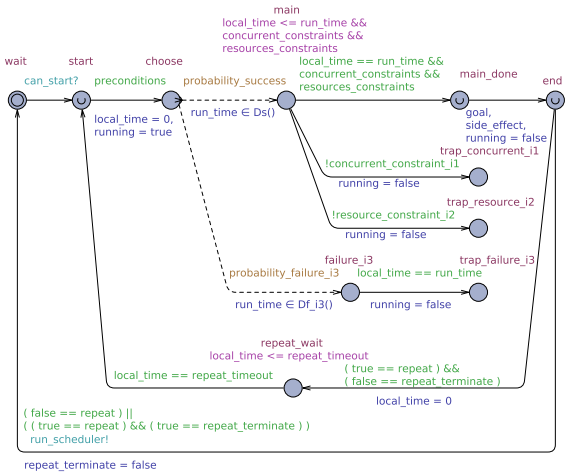

Figure 2: Template for PTA_{PLP} achieve

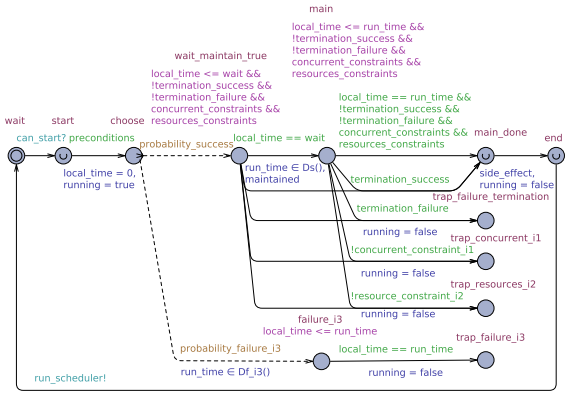

Figure 3: Template for PTA_{PLP} maintain

sition 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.

Logical conditions in the PLP are converted to guard conditions in the PTA above 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 $i_1 \in \left[1, m_1\right]$ 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 $i_2 \in \left[1, m_2\right]$, 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 boolean variable *"repeat"*.

Assignment conditions in the PLP are converted to assignments in the PTA above as follows: 1. The transition between *"main_done"* to *"end"* takes care of updating 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.

Maintain

PTA_{PLP} *maintain*, shown above in Figure 3. is similar to PTA_{PLP} *achieve* with a few changes: 1. An additional node *"wait_maintain_true"* that models the time needed by PLP Maintain for the maintained condition to become true. 2. The condition maintained by the PLP is converted to an assign-

ment (to model the change caused by the underlying module). This assignment appears here as *"maintained"*, which is an update on a transition from *"wait_maintain_true"* to *"main"* node. 3. Successful termination of the PLP is accomplished by the *"termination_success"* condition that can force a transition to *"main_done"*. 4. Unsuccessful termination is accomplished by the *"termination_failure"* condition that can force a transition to *"trap_failure_termination"*.

In the template above, the PTA_{PLP} maintain implements the maintained condition (i.e., forces it to be *true*) using the *"maintained"* assignment. This is done only once, before *"main"* node. It is also possible to ensure that the assignments in *"maintained"* are not overwritten with other values during the execution of the PTA. This can be accomplished by converting the *"maintained"* assignments to a logical condition in the invariant of the *"main"* node, and creating a failure state in case it changes while in *"main"*.

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, 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. See Kovalchuk (2018) for additional details.

Control Graphs

We use *control graphs* to describe algorithms controlling execution of robotic modules specified by PLPs. They allow for probabilistic and conditional branching, as well as parallel execution.

A control graph is a directed graph with a single root node in which execution starts. The nodes of the control graph correspond to 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. 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.

There are four types of control nodes: 1. PTA_{PLP} launcher node – launches sequence of PTA_{PLP} that execute one at a time. 2. Probabilistic node – chooses 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 node – chooses a single edge to proceed with. Only an edge whose condition is satisfied can be selected. If more than edge condition is satisfied, one is selected nondeterministically. (If no condition is satisfied, then this is viewed as a failure.) 4. Concurrent node – executes all the outdoing edges concurrently.

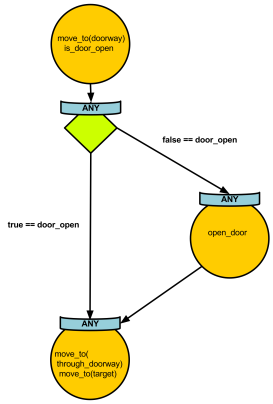

Figure 4: Example of a control graph

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

Nodes in the control graph can update the value of variables that are used both by PTAs_{PLP} and other nodes in the control graph.

Figure 4 illustrates a control graph for an autonomous robot with an arm whose task is to move the robot from one room to another through a doorway. The door in not locked and can be either open or closed. The control graph refers to three PTAs_{PLP}: 1. move_to(doorway | target | through_doorway) – moves the robot according to its parameter. 2. is_door_open – checks if the door is open. 3. open_door – opens the door. This control graph starts with a node that executes move-to-doorway and checks if the door is open. Then, it uses a conditional node to check if the door should be opened. In case the door is closed, it opens it. Eventually, it moves through the doorway to the

target at the other room.

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.

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 be eventually true? 7. Probability of an invariant: What is the probability that p will always be true?

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 mapping is quite technical, and its details appear in Kovalchuk (2018).

Empirical Evaluation

We evaluate the performance and scalability of this approach, as implemented in our verification software, available at https://github.com/a-l-e-x-d-s-9/plps_verification. We evaluate resource demands of the system in two phases: The first phase is the compilation of a control graph and PLPs for UPPAAL. We expect the compilation time will grow as with more variables, complex conditions, complex and larger PLPs and control graph. The second phase is verification of queries on an already compiled system in UPPAAL. The querying phase is performed on a single file that contains the PTAs that correspond to all the initial PLPs and control nodes. With larger and more complex graphs of PTAs, we expect UPPAAL will require more time and more memory to answer queries.

To evaluate the practical restrictions of the system in both phases we use two independent test

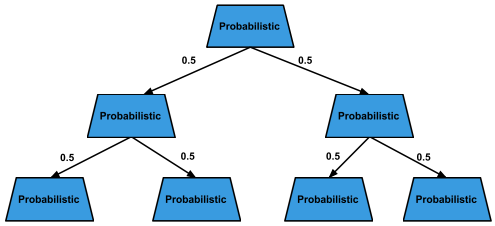

Figure 5: First Part of First Control Graph

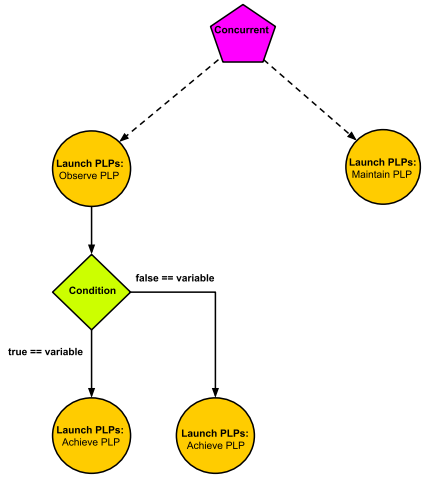

Figure 6: Second Part of First Control Graph

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 of this system is composed of two conceptual parts: The first part is a full binary tree of probabilistic nodes. Our generator takes the desired height of the tree, and generates a control graph of the form shown in Figure 5, whose number leaf nodes is exponential on its height.

Each leaf node is associated with an independent control sub-graph shown in Figure 6. The root node in this sub-graph 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

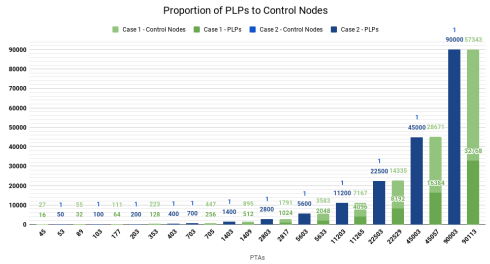

Figure 7: Proportion of PLPs to Control Nodes

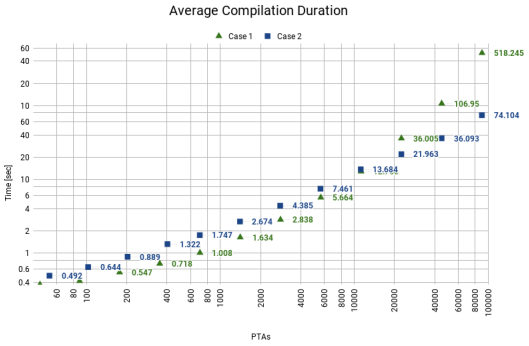

Figure 8: Average Compilation Time

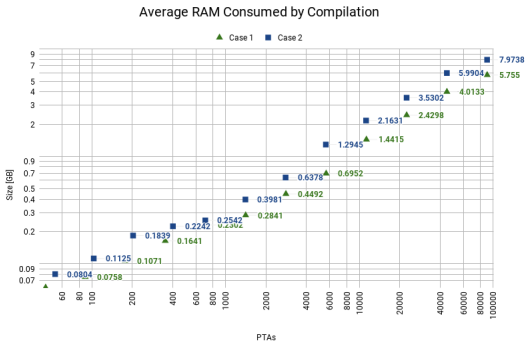

Figure 9: Average RAM Consumed by Compilation

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 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 $\times$ 8 with 16 GB RAM, SSD, Java 1.8.0 171, and Ubuntu 17.10 64bit. We used the latest 64bit version of UPPAAL (specifically "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 is 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. The proportions between PLPs and *control nodes* vary between the two test cases and are shown in Figure 7.

We tested the compilation process with up to 90,000 PTAs. Figures 8 and 9 describe its run-time and memory consumption for both test cases as a function of total amount of PTAs in a system (axes are logarithmic). These results clearly indicate that the compilation process, which is a one-time process, is quick and scales to very large problems. In fact, we cannot envision, in the near future, a system with more than a few dozen components, hence compilation will not be a bottleneck.

The compilation process produces a single file for UPPAAL. As can be seen in Figure 10, even a system with 90,000 PTAs produces files of size smaller than 1GB.

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

The time and memory consumed by the query are shown in Figures 11 and 12. Query cost does

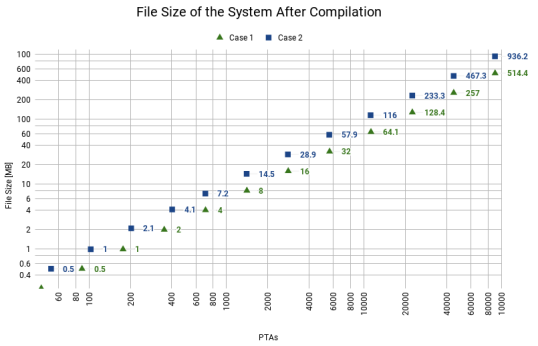

Figure 10: File Size Following Compilation

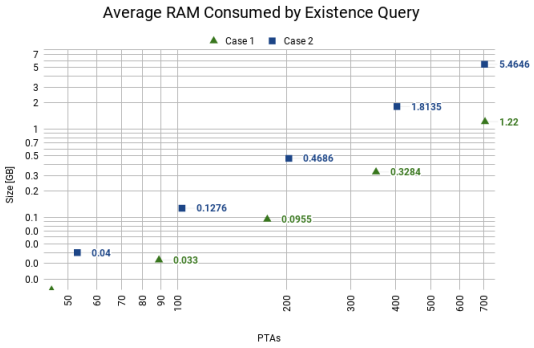

Figure 12: Avg. RAM Consumed for ∃ Query

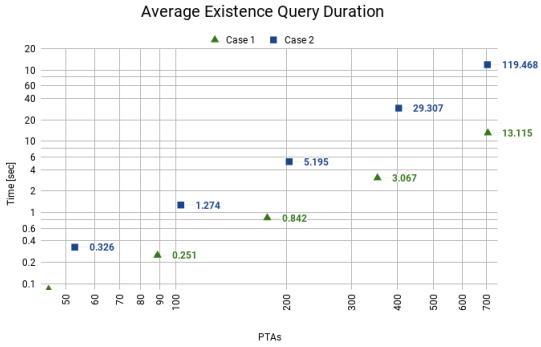

Figure 11: Average Time for ∃ Query

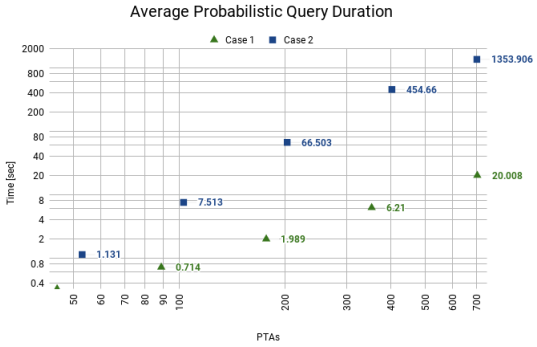

Figure 13: Average Time for Probability Query

not scale up as well as compilation cost. In particular, systems with over 700 PTAs cause UPPAAL to crash due to stack overflow. However, for moderate system sizes, it is relatively efficient, and multiple queries can be carried out in reasonable time. In fact, in system with less than 200 PTAs, online queries for evaluating plans can be supported.

The second query we tested was a probability ("Pr <>") query of reaching successfully 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 Figures 13 and 14. 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.

Summary and Future Work

We described a formal semantics for *Performance Level Profiles* (PLPs) by mapping them to probabilistic timed automata (PTAs). Because PLPs have a formal syntax, they are machine readable and processable, allowing us to leverage this semantics to actually map actual PLPs to PTA specifications. We then extended this mapping to compositions of PLPs, enabling us to capture complex control structures that use concurrency, conditions, loops, and randomization. This allows us to map complex scripts that invoke code fragments for which a PLP exists, into interacting PTAs. By feeding the resulting PTAs into a verification engine, we can verify various properties of the script,

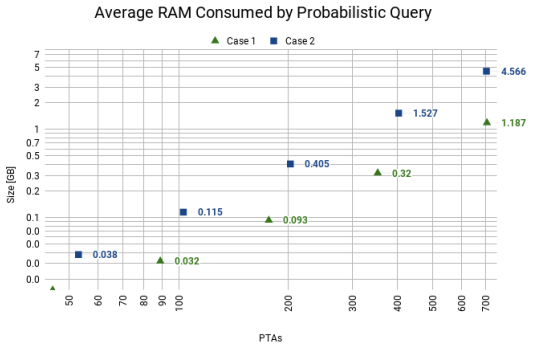

Figure 14: Avg. RAM Consumed for Prob. Query

such as the probability it will achieve certain conditions, the probability it will maintain an invariant, and more. Our empirical evaluation indicates that this approach is quite scalable.

Acknowledgements

We thank the reviewers for their useful comments. This work was supported by ISF Grants 1651/19, 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 *2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, May 6-10, 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)*.

Ashkenazi, M. 2017. Planning and monitoring with performance level profles. Master's thesis, Ben-Gurion University.

Beauquier, D. 2003. On probabilistic timed automata. *Theor. Comput. Sci.* 292(1):65–84.

Behrmann, G.; David, A.; Larsen, K. G.; Hakansson, J.; Petterson, P.; Yi, W.; and Hendriks, M. 2006. Uppaal 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, 125–126. IEEE.

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 *Proceedings of International Conference on Intelligent Robots and Systems*.

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 Brinksma, E., and Larsen, K. G., eds., *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, 359–364. Springer.

Foughali, M.; Ingrand, F.; and Seceleanu, C. 2019. Statistical model checking of complex robotic systems. In Biondi, F.; Given-Wilson, T.; and Legay, A., eds., *Model Checking Software - 26th International Symposium, SPIN 2019, Beijing, China, July 15-16, 2019, Proceedings*, volume 11636 of *Lecture Notes in Computer Science*, 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, Wisla, Poland, 20-22 October 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.

Ingrand, F.; Catilla, R.; Alami, R.; and Robert, F. 1996. A high level supervision and control lan-

guage 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, HASE 2012, Omaha, NE, USA, October 25-27, 2012*, 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., and Pommereau, F. 2018. Aspic: An acting system based on skill petri net composition. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2018, Madrid, Spain, October 1-5, 2018*, 6952–6958. IEEE.

Lesire, C.; Doose, D.; and Grand, C. 2020. Formalization of robot skills with descriptive and operational models. In *IROS'20*.

Mallet, A.; Fleury, S.; and Bruyninckx, H. 2002. A specification of generic robotics software components: future evolutions of g$^{\text{en}}$$_{\text{o}}$m in the orocos context. In *IEEE IROS*, 2292–2297.

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 *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2000, October 30 - Novemver 5, 2000, Takamatsu, Japan*, 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.