A Formal Semantics for PLPs using Probabilistic Timed Automata, and its Application to Controller Verification using UPPAAL

Thesis by

Alex Kovalchuk
alex.cs.rnd@gmail.com

Supervised by

Ronen I. Brafman
brafman@cs.bgu.ac.il

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Computer Science

Ben-Gurion University of the Negev
Beer Sheva, Israel

October 2017

Abstract

Performance Level Profiles (PLPs) were developed as a formal, quantitative language for specifying the behavior of functional modules. There are four types of PLPs: 1. Achieve - achieves internal and external goals. 2. Observe - senses environment. 3. Maintain - maintains a property. 4. Detect - detects property. In this thesis, we propose a formal way to represent all the PLPs in terms of *Probabilistic Timed Automatons* (PTAs). To go beyond single module level we define a *control graph* as a formal representation of execution plan for robotic modules specified by PLPs, enhanced with probabilities, conditional branching, parallel execution etc. Finally, we developed software that compiles PLPs specification and *control graph* plan into a single system suitable for offline analysis by UPPAAL model checker.

Contents

CHAPTER 1

Introduction

In recent years, we are witnessing accelerated development in the field of autonomous robotic systems. There are many possible applications for autonomous robots, and each application introduces new challenges. Especially challenging is the development of autonomous robots functioning alongside humans in a safe and reliable manner. Hence the growing need for methods for building reliable and predictable autonomous robots. Currently, there are a few existing tools [3] that seek to achieve these goals by defining formal programming languages for describing a robotic system that generates code with various guarantees and support the ability to prove and infer system properties. Unfortunately, these tools have not been widely adopted, and most developers prefer to use regular programming languages to design robotic components.

One of the key problems with code written using standard programming languages is that it is not always clear what level of performance one can expect from it. Such information is required to understand what one can expect from a robot using such code. It is essential information for predicting its behavior and is required to enable appropriate reuse of code. This information can sometimes be obtained from its documentation, but the standard specification techniques used by most software engineers are informal, qualitative, and use free text which is not machine readable.

Performance Level Profiles (PLPs) [6] [5] were developed as a formal, quantitative language for specifying the behavior of functional modules. PLPs were developed to help developers who use regular programming languages provide a formal, quantitative, machine readable specification of their module's behavior. This information makes it possible to reuse the abundant existing modules more reliably and also makes possible automatic analysis, monitoring of modules, as well the use of automated learning techniques to update their descriptions. PLPs can be used by a robotic control unit to provide it with information about conditions needed to execute a module, conditions made true by the module when done successfully, possible failure modes, the module's probability to complete its work successfully without any failure, time distribution of its execution, and more.

There are four types of PLPs: 1. Achieve - works to achieve some desirable property, internal or external. 2. Observe - attempts to reveal a value of an environmental variable. 3. Maintain - works to keep a certain variable in particular range or maintain some condi-

tion while running. 4. Detect - attempts to identify some condition that is either not true now, or that is not immediately observable.

Generally, PLPs provide a basis to look at a software module as a sort of logical unit that can be executed only if its preconditions are satisfied, and when executed it achieves certain goals. Elements used by PLPs can provide a foundation for modular system design and other complex applications. Currently, software tools for PLPs [5] are used to: 1. Generate code automatically from a PLP to monitor conditions specified by the PLP 2. Analyze performance to update the PLP specification. 3. Generate plans that achieve user specified tasks on the fly, by both generating domains descriptions that can be used in off-the-shelf planners, and also providing the needed software interfaces between the planner and the underlying functional module.

But while PLPs are quite intuitive, they lack a formal semantics. Such semantics is essential to make their meaning clear and is required to be able to prove properties. Moreover, a robot's behavior is the result of the execution of multiple modules combined in diverse ways, and one would want to go beyond the module level in order to characterize and prove properties of the entire program. The goal of this thesis is to address these problems theoretically and practically.

In this thesis, we propose a formal semantics for PLP defined by *Probabilistic Timed Automaton* (PTA) [18], on which we apply then an existing model checking tool. We introduce a structure, *control graph*, which represents a control unit of a robot that uses PLPs as functional units. We show how *control graphs* and PLPs can be used to verify properties of the whole system offline, and we provide a software tool implementing this technique.

Any formal model of PLPs must incorporate some elements that are an essential part of PLPs. More specifically, we want a model that can represent and use: numbers, numerical variables, numerical comparisons, logical conditions, time progress, distribution of run time, and probabilistic transitions and events. The simplest model we found that captures these aspects while providing sufficiently mature model checking tool and clear semantics is *Probabilistic Timed Automaton* (PTA) [18].

A *Probabilistic Timed Automaton* (PTA) is a type of finite hybrid automata extended with probabilities. PTA uses real-valued variables called *clocks* that increase simultaneously at the same rate. PTA transitions can contain logical conditions on boolean or *real* variable, *clocks* comparison to *real* values, and branch points with probabilities on each transition. With a PTA description of a system, we can ask certain types of questions about the system. Typical questions are: 1. Reachability - whether a certain state could be reached by some valid execution path. 2. Safety - whether certain condition always holds. 3. Liveness - whether a specific condition holds eventually. 4. Deadlock - is deadlock possible or not. All the questions above can be addressed with model checking algorithms on a PTA model. We use UPPAAL [19] [8] as our model checking tool, due to its support of all the essential features.

We use PTA qualities (logical conditions, *clocks,* and probabilities) to create a single PTA template for each PLP type thus providing it with a semantic model. Each specific PLP instance of each type will be represented by an automaton with unique conditions, effects, time distributions, probabilities and goals, which constitute an instantiation of the general PTA schema for the corresponding PLP type.

But we want to go beyond the level of a single functional module. We would like to be able to use the above semantics to verify the properties of complex behaviors that involve multiple functional modules executed in complex ways. But if we want to address bigger problems that require complex robotic behavior, dealing with individual models is not enough. Usually, such behavior is not a mere sequence of execution of modules, but an algorithm or program that we refer to as a robotic controller, or controller, for short. A controller controls the robot and schedules the execution of particular modules. We assume that the controller can access robot's sensors and more generally any robotic module attached to it. Such a controller is usually designed with some goal in mind.

Usually, a controller works directly with specific robotic modules through a custom interface and without a standard technique for analysis, monitoring, and logging. This practice limits the opportunities for reusability of modules and automation of the system. We can address these limitations by using a PLP based interface between the controller and actual robotic module. We will define a *control graph* as a general description of a robotic controller that will use a PLP based interface to control robotic modules.

The *control graph* describes controller algorithms. It is very similar to decision trees in structure, enhanced with multiple execution paths and nodes that function as intersections of execution paths that fuse into a single path. *Control graphs* can use variables shared with PTAs representing certain PLPs, to provide input arguments to PTAs and to receive outputs from them. The *control graph* uses conditions and probabilities to determine the execution sequence of PTAs representing PLPs. A *control graph* is built from 4 elementary nodes: 1. *Sequential node* - executes a sequence of PLPs. 2. *Probabilistic node* - has multiple outgoing edges, each with probability to be executed. 3. *Conditional node* - outgoing edges have conditions, and only a single edge with a true condition can be executed. 4. *Concurrent node* - all the outgoing edges are scheduled at the same time and executed concurrently.

Control graphs and PLPs are the inputs to our software tool. Each node of a *control graph* is compiled into a PTA: 1. A *sequential node* schedules the execution of PLPs by the defined sequence, waiting for each PLP to finish. 2. Probabilistic and *conditional nodes* use PTAs' edges with probabilities and conditions. 3. *Concurrent node* schedules execution of multiple subsequent nodes, according to the outgoing edges. After the compilation of *control graph* and PLPs to a network of PTAs, we use the UPPAAL model checker to query the whole system. The designer of a controller could use results of our compilation as a way to verify that the system would work as he expects it to work. In particularly the designer can verify that certain states are reachable, and estimate whether

the probabilities to reach those states are acceptable. Conclude the final goal of this thesis.

Contributions:

- An introduction of formal semantics for PLPs with *Probabilistic Timed Automaton* templates representation for each PLP type. This semantics capture the aspects of PLPs: logical premises, probabilistic outcomes, and temporal distributions. Furthermore, we introduce a mechanism for the enforcement of *concurrent conditions*.
- An introduction and formalization of *control graph* as a relatively simple way to represent robotic controllers and algorithms, based on robotic modules specified by PLPs.
- A verification method of a system combining *control graphs* and PLPs by conversion of the system to a network of PTAs and using a model checker on it. Including working implementation in Java.

CHAPTER 2

Background

2.1 *Performance Level Profiles*

Performance Level Profiles serve as agreements between robotic module designers and systems that use it. PLPs describe the conditions under which a module should be operated, its expected run time and probabilities of success and failures. PLPs were designed with rigid and machine-readable syntax, with the following objectives in mind: 1. Automated analysis, monitoring, and reuse of existing code. 2. Convey to customers a certain expected level of performance of an autonomous robot. 3. Quickly identify an abnormal behavior of autonomous robots. Currently, there are four types of PLPs: Achieve, Maintain, Observe and Detect. They share common characterizations but differ in the purposes of the modules they represent.

2.1.1 Common elements

- Parameters - input and output variables used by the module.
- Required resources - list of resources and quantities (if needed).
- Preconditions - conditions involving parameters that must be satisfied for the module to be executed.
- *Concurrent conditions* - conditions on variables that must be satisfied while the current module is being executed.
- *Concurrent module*s - conditions on other modules that must or must not be executed at execution time of the current module.
- Parameter frequency (optional) - the frequency of reading or writing of variables.
- Side-effects - unintended effect of a running module, described by a conditional assignment to a parameter. For example consumption of power resource.
- Expected progress rate - expected rate of completing the task.

2.1.2 *PLP Achieve*

Achieve modules want to accomplish some desirable property. That can be either an external state of the world or internal virtual property. An example for an external goal for Achieve module is to reach world state in which the fuel tank is full. An example of an internal Achieve module is to generate a path to a certain destination.
Additional elements:

- Goal - a boolean condition that is achieved by the module.
- Failure modes - possible failure cases.
- Success probability.
- Failure modes probability.
- Run time distribution - for success and failure modes.

2.1.3 *PLP Maintain*

Maintain modules keep the value of a variable at a certain value or the truth value of some condition. The maintained conditions can be false at the beginning, and it may need some time to become true. Maintain module can be used for example to maintain a certain speed or maintain direction.
Additional elements:

- Maintained condition - condition to be maintained by the module.
- Whether the maintained condition is initially true.
- *Termination condition* for success.
- *Termination conditions* for failure modes.
- Success probability.
- Failure modes probability.
- Run time distribution - for success and failure modes.

2.1.4 *PLP Observe*

Observe type of module attempt to reveal the value of some variable or boolean condition. For example, observe whether the robot is standing.
Additional elements:

- Observation goal - boolean condition to be verified or a parameter whose value is to be observed.
- Success probability.
- Failure to observe probability.
- Run time distribution - for success and failure modes.

2.1.5 *PLP Detect*

Detect modules try to identify some condition that is either not true now, or that is not immediately observable. For example, intruder detection or detect an obstacle.
Additional elements:

- Detect goal - the condition being detected.
- Success probability.

2.1.6 Repeat

Repeat is an extension made for modules that should be run continuously, to repeat a certain task. Repeat can be used with *PLP Achieve* and *PLP Observe* that should be executed repeatedly in certain frequency until its *termination condition* becomes true. Instead of hiding the repetitions inside the module, repeat make possible to control the repetition parameters: 1. *Execution frequency* - how many times the module is executed per unit time. 2. Input frequency - minimal expected update frequency of variables. 3. Repeat *termination condition* - the condition that stops the repeat of a module.

2.2 Probabilistic Timed Automata (PTAs)

Probabilistic timed automata (PTAs) are a formalism for modeling systems whose behavior incorporates both probabilistic and real-time characteristics. [18] [16]

2.2.1 Time and *clocks*

Probabilistic timed automata model time using *clocks*, variables over the set $\mathbb{R}_{\geqslant 0}$ of non-negative *reals*. We assume a finite set $\mathscr{X}$ of *clocks*. A function $v : \mathscr{X} \to \mathbb{R}_{\geqslant 0}$ is referred to as a *clock valuation* and the set of all *clock* valuations is denoted by $\mathbb{R}_{\geqslant 0}^{\mathscr{X}}$. For any $v \in \mathbb{R}_{\geqslant 0}^{\mathscr{X}}$, $t \in \mathbb{R}_{\geqslant 0}$ and $X \subseteq \mathscr{X}$, we use $v+t$ to denote the *clock* valuation which increments all *clock* values in v by t and $v[X := 0]$ for the valuation in which *clocks* in X are reset to 0.
The set of *clock* constraints over $\mathscr{X}$, denoted $CC(\mathscr{X})$, is defined by the syntax:

$$\chi ::= true \mid \text{x} \leqslant d \mid \text{c} \leqslant x \mid \text{x+c} \leqslant y + d \mid \neg\chi \mid \chi \wedge \chi$$

where $x, y \in \mathscr{X}$ and $c, d \in \mathbb{R}$. A *clock* valuation v satisfies a *clock* constraint χ, denoted by $v \vDash \chi$, if χ resolves to true when substituting each occurrence of *clock* x with $v(x)$. *Clock* constraints will be used in the syntactic definition of PTAs and for the specification of properties.

2.2.2 PTA Syntax

A *Probabilistic Timed Automaton* (PTA) is a tuple $P = (L, l_0, \mathscr{X}, Act, inv, enab, prob)$ where:
– L is a finite set of locations and $l_0 \in L$ is an *initial location*;
– $\mathscr{X}$ is a finite set of *clocks*;
– Act is a finite set of *actions*;
– $inv : L \to CC(\mathscr{X})$ is an *invariant condition*;
– $enab : L \times Act \to CC(\mathscr{X})$ is an *enabling condition*;
– $prob : L \times Act \to Dist(2^{\mathscr{X}} \times L)$ is a (partial) *probabilistic transition function*.

A state of a PTA is a pair $(l, v) \in L \times \mathbb{R}^{\mathscr{X}}_{\geqslant 0}$ such that $v \vDash inv(l)$. In any state (l, v), either a certain amount of time $t \in \mathbb{R}_{\geqslant 0}$ elapses or an action $a \in Act$ is performed. If time elapses, then the choice of t requires that the invariant $inv(l)$ remains continuously satisfied while time passes. Each action a can be only chosen if it is enabled, that is, the condition $enab(l, a)$ is satisfied by $v+t$. Once action a is chosen, a set of *clocks* to reset and successor location are selected at random, according to the distribution $prob(l, a)$. We call each element $(X, l') \in 2^{\mathscr{X}} \times L$ in the support of $prob(l, a)$ an edge and, for convenience, assume that the set of such edges, denoted $edges(l, a)$, is an ordered list $\langle e_1, \ldots, e_n \rangle$.

We assume that PTAs are well-formed, meaning that, for each state (l, v) and action a such that v satisfies $enab(l, a)$, every edge $(X, l') \in edges(l, a)$ results in a transition to a valid state, i.e., we have $v[X := 0] \vDash inv(l')$. A PTA can be transformed into one that is well-formed by incorporating the invariant associated with the target location into the *enabling condition* of each location-action pair [15].

2.2.3 PTA Semantics

Let $P = (L, l_0, \mathscr{X}, Act, inv, enab, prob)$ be a PTA. The semantics of P is defined as a tuple $(S, s_0, Act \times \mathbb{R}_{\geqslant 0}, Steps_P)$ where:
– A probability distribution function μ.
– Set of states $S = \left\{(l, v) \in L \times \mathbb{R}^{\mathscr{X}}_{\geqslant 0} \mid v \vDash inv(l)\right\}$ and initial state $s_0 = (l_0, \mathbf{0})$;
– for any $(l, v) \in S$ and $a \in Act \cup \mathbb{R}_{\geqslant 0}$, we have $Steps_P((l, v), a) = \lambda$ if and only if either:
Time transitions. $a \in \mathbb{R}_{\geqslant 0}, v + t' \vDash inv(l)$ for all $0 \leqslant t' \leqslant a$ and $\lambda = \mu_{(l, v+a)}$;
Action transitions. $a \in Act, v \vDash enab(l, a)$ and for each $(l', v') \in S$:

$$\lambda(l', v') = \sum \left\{ \mid prob(l, a)(X, l') \mid X \in 2^{\mathscr{X}} \wedge v' = v[X := 0] \mid \right\}.$$

2.3 PTA augmentations

We now summarize a variety of extensions to the standard definition of PTAs that facilitate high-level modeling using this formalism.

2.3.1 Parallel composition

It is often useful to define complex systems as the *parallel composition* of several interacting components. The definition of the *parallel composition* operator $\|$ for PTAs [14] uses ideas from (untimed) probabilistic automata [22] and classical timed automata [1]. Let $P_i = (L_i, \bar{l}_i, \mathscr{X}_i, Act_i, inv_i, enab_i, prob_i)$ for $i \in 1, 2$ and assume that $\mathscr{X}_1 \cap \mathscr{X}_2 = \emptyset$. Given $\mu_1 \in Dist(2^{\mathscr{X}_1} \times L_1)$ and $\mu_2 \in Dist(2^{\mathscr{X}_2} \times L_2)$, we let $\mu_1 \otimes \mu_2 \in Dist(2^{\mathscr{X}_1 \cup \mathscr{X}_2} \times (L_1 \times L_2))$ be such that $\mu_1 \otimes \mu_2(X_1 \cup X_2, (l_1, l_2)) = \mu_1(X_1, l_1) \cdot \mu_2(X_2, l_2)$ for $X_i \subseteq \mathscr{X}_i, l_i \in L_i$ and $i \in \{1, 2\}$. The *parallel composition* of PTAs P_1 and P_2 is the PTA:

$$P_1 \parallel P_2 = (L_1 \times L_2, (\bar{l}_1, \bar{l}_2), \mathscr{X}_1 \cup \mathscr{X}_2, Act_1 \cup Act_2, inv, enab, prob)$$

such that, for each location pair $(l_1, l_2) \in L_1 \times L_2$ and action $a \in Act_1 \cup Act_2$:

- the *invariant condition* is given by $inv(l_1, l_2) = inv_1(l_1) \wedge inv_2(l_2)$;
- the *enabling condition* is given by:

$$enab((l_1, l_2), a) = \begin{cases} enab_1(l_1, a) \wedge enab_2(l_2, a) & \text{if } a \in Act_1 \cap Act_2 \\ enab_1(l_1, a) & \text{if } a \in Act_1 \setminus Act_2 \\ enab_2(l_2, a) & \text{if } a \in Act_2 \setminus Act_1 \end{cases}$$

- the *probabilistic transition function* is given by:

$$prob((l_1, l_2), a) = \begin{cases} prob_1(l_1, a) \otimes prob_2(l_2, a) & \text{if } a \in Act_1 \cap Act_2 \\ prob_1(l_1, a) \otimes \mu_{(\emptyset, l_2)} & \text{if } a \in Act_1 \setminus Act_2 \\ \mu_{(\emptyset, l_1)} \otimes prob_2(l_2, a) & \text{if } a \in Act_2 \setminus Act_1 \end{cases}$$

2.3.2 Variables

When modeling systems with probabilistic timed automata, it is often convenient to augment the model with discrete variables [4] [25]. For the sake of simplification of PLP to PTA conversion, we will use a stronger augmentation with an addition of real-valued variables. *Enabling conditions* can then refer to the current value of the variables, and the probabilistic transition relation is extended to allow updating variable values. We add to the PTA definition a finite set of variables U , each variable can be evaluated with valuation function $\forall u \in U : v(u) \in \mathbb{R}$.

Enabling condition is extended to contain *variable constraints* additionally to *clock* constraints $CC(\mathscr{X}) \wedge VC(U)$. $VC(U)$ is defined by the syntax:

$$\omega ::= true | \text{ x} \leqslant d | \text{ c} \leqslant x | \text{ x+c} \leqslant y + d | \neg\omega | \omega \wedge \omega$$

where $x, y \in U$ and $c, d \in R$.

*Probabilistic transition function*s are extended to include value update for a subset of variables. Variable is either updated to a result of a function depending on the subset of variables, or a probabilistic value corresponding to some distribution.

2.3.3 Urgency

When modeling real-time systems, it is often necessary to express the fact that a particular action should be taken immediately, without letting time pass. In this way, we can model, for example, an instantaneous system event comprising several atomic actions. A few mechanisms for modeling such situations have been introduced for timed automata, for example in the system-description language of the UPPAAL model checker [4]; here, we describe how they are adapted to PTAs.

An *urgent* location of a PTA is a location in which no time can pass. *Urgent* locations can be represented in the PTA framework by introducing an additional *clock*, which is reset on entry to an *urgent* location, and by including a conjunct in the *invariant condition* of the location to specify that the value of the *clock* should be equal to 0 in the location.

2.3.4 Channels

Channels are used to synchronize between two or more PTAs in a single *parallel composition*.
The definitions of actions and *parallel composition* can be extended to allow sending and receiving (to either single or multiple recipients) of messages along with them, as in UPPAAL [4]. Such behavior can be encoded in the action names of a standard PTA.

2.4 PTA representation as graph

PTAs can be represented in a much more compact and human-friendly form by a graphical representation, which preserves all the details of PTA semantics. Any instance of a PTA can be converted to unique PTA graph, and any PTA graph can be converted back to a PTA.
We will present how exactly the conversion from PTA semantic to PTA graph made and then we will be using a graphical representation for PTA.

For a PTA $P = (L, l_0, \mathscr{X}, U, Act, inv, enab, prob)$:

– L - *locations* - each location represented by a node, with the same name. Initial location represented by a double outline on a single node. Time can pass only inside node;

– $\mathscr{X}$ - *Clocks* - unchanged;

– Act - *actions* - each action represented by a directed edge connecting two nodes, drawn as arrows;

– inv - *invariant condition* - represented as a condition located in attribute *"invariant"* attached to a node. *Invariant* can be omitted if true;

– $enab$ - *enabling condition* - represented by a condition located in attribute *"guard"* attached to an edge. Guard can be omitted if true. Additionally to conditions involving *clocks*, guard can also contain conditions involving variables;

– $prob$ - *probabilistic transition function* - represented by a number in a range [0,1] located in attribute *"probability"* attached to an edge. *Probability* can be omitted if equal for all outgoing edges. Edges with omitted probabilities drawn by an arrow with continuous line, edges with non-omitted probabilities drawn by arrows with dashed lines.

– *Clocks* resets and change of values for variables located in attribute *"update"* attached to an edge. *Update* is executed when action transition for a specific edge is being executed. Variable update with a probability according to a certain distribution would appear as a pattern: $variable\_name \in distribution()$.

– *Urgent* locations - will be represented by "U" mark inside a node.

– Channels will be represented by a label with "?" or "!" sign located in attribute *"synchronization"* attached to an edge. "!" - used by PTA sending a *synchronization signal*. "?" - used by PTA receiving *synchronization signal*.

– Run time distribution - will be part of the *guard* attribute, it would appear as pattern: $clock\_name \in distribution()$.

2.5 PTA example

Here we have a simple communication protocol represented with PTA. The PTA has two *clocks* *x* and *y*, which start with the value 0. In the location *start* , the system waits for at least 1 time-unit (represented by the *enabling condition* $x >= 1$ on the outgoing distribution of action send) and at most 2 time-units (represented by $x <= 2$ in the *invariant condition*), before sending a message. With probability 0.9 the message is received correctly (edge to *done*); otherwise, with probability 0.1, the message is lost (edge to *lost*). In the latter case, once *clock x* reaches 8, the PTA returns to *start* where another attempt to send the message can be made. If in total, at least 20 and at most 25 time-units have elapsed since the start of system execution, the PTA performs a timeout and moves to location *fail* .

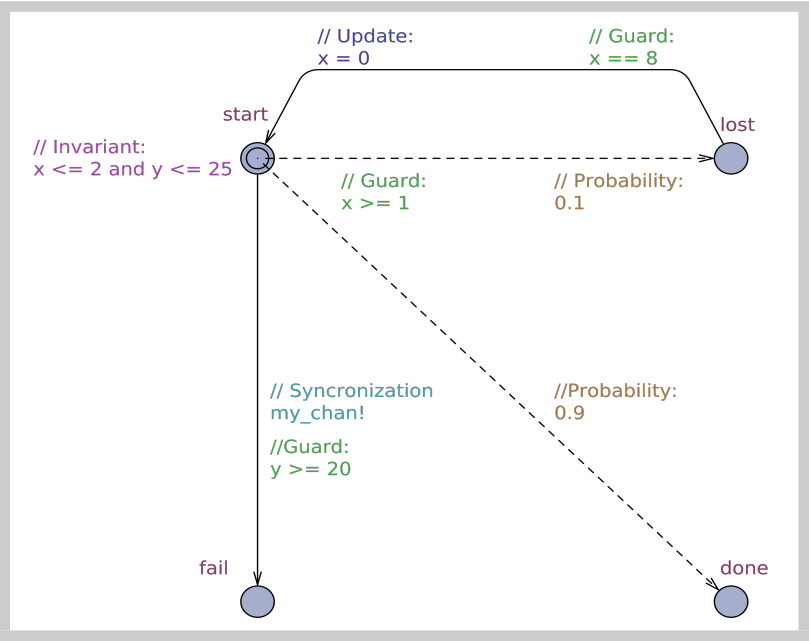


FIGURE 2.1

In the future graphical representations of PTAs, we will not be writing the role of each attribute; they will be close to node or edge they belong to, and color-coded:

Attribute	RGB(hexadecimal #RRGGBB)	Color
Node Name	#8C3863	
Node Invariant	#A742A8	
Edge Guard	#42A848	
Edge Probability	#A87A42	
Edge Update	#4242A8	
Edge Synchronization	#42A0A8	

CHAPTER 3

Foundation for Formal Semantics for PLPs

We saw that there are four types of PLPs that describe four prototypes of robotic modules: Achieve, Maintain, Observe and Detect. All PLPs capture the three possible stages of robotic module execution: 1. Before execution - the parameters, resources, and conditions needed to run the module. 2. While executing - *concurrent conditions*, and side effects. 3. After execution - a probability of success and failure, run time, and goal. We want to go over the specification of each type of PLP and create a representative PTA template, i.e., PTA_{PLP} that would embody PLP's qualities such as run time distribution, a probability of success and failure, values changes of variables in a *goal condition*, etc. PTA_{PLP} is defined solely based on the information specificated in a PLP - the real underlying module, whether exists or not, is never used in the specification. In this section, we lay the foundation needed for PTA_{PLP} definitions. We will go through the individual components we use to transpose PLP into PTA_{PLP}.

3.1 Variables

All types of variables used by PLP, namely: parameters, variables, constants, and required resources are converted to real-valued variables of PTA. For boolean variables, false is defined as 0, and true as 1.

3.2 Conditions

The term *condition* is used in two different meanings in PLP: 1. The way we usually think of condition, for example, a PLP precondition: *"a = b"* - meaning that if the value of *a* equals value of *b* then *"a = b"* is true, but *a* and *b* do not have to be equal and the condition can be false. 2. As a statement that needs to be true and requires the variables to be adjusted accordingly. That is, as an indirect assignment statement. For example, the PLP *goal condition*: *"a = b"* means that it must be true: Whatever the values of *a* and *b* were before; they must be equal to the same value at the end. Basically, the first meaning is a condition and the second meaning is an assignment of value.

In the first case, to represent regular conditions, we use *guard* attribute of a certain edge. *Guard* attribute can contain: logical AND operators, logical NOT operators, comparison operator *equal or smaller* (<=) on *clocks* and *integers*. Comparison operator *equal* (==) can be represented by: *a == b iff (a <= b) AND (b <= a)*. Comparison operator *equal or larger* (>=) can be represented by: *a >= b iff (b <= a)*. Comparison operator *larger* (>) can be represented by: *a > b iff (b < a)*. Comparison operator *smaller (<)* can be represented by: *a < b iff (a <= b) AND NOT(a == b)*. Logical OR operator can be represented by using 2 edges, each one with a guard representing a different operand of logical OR.
To conclude, for a regular type of conditions in PLP we go over the conditions, possibly recursively, and produce the corresponding PTA_{PLP} *guard*.

In the second case, condition as an assignment, we use an edge's *update* attribute, to assign a value to variables. The condition from PLP is translated to a series of assignments of values to variables. The condition of type *"v == C"*, where *v* is variable and *C* is constant, translated into assignment *"v ← C"*. The condition of type *"(C_1 <= v) AND (v <= C_2)"*, where v is variable and C_1, C_2 are constants, should produce the assignments: *"v ← C_1, v ← C_1+1, …, v ← C_2 - 1, v ← C_2"*, but assignments for the same variable with different values are meaningless if done in *update* of single edge, therefore it will be split into multiple edges, each edge with single assignment into variable *v* in its *update*. Similarly, more complex conditions can be converted eventually into a set of assignments.

3.3 Resources

PLP can require certain resources to run, and optionally a minimum amount of them. We will define a single variable for each resource. PTA_{PLP} will check that there is enough of the resource. If an exclusive access to a resource required, we will check that it is indeed accessed by a single PLP.

3.4 Preconditions

Preconditions are conditions on variables required by PLP to run. We convert all the PLP type preconditions into PTA type conditions, then concentrate them as part of a *guard* attribute at the beginning of every PTA_{PLP}.

3.5 Goal

The goal is accomplished when PLP is done successfully. We convert PLP goal into series of assignments into variables, at the end of successful completion of PTA_{PLP}.

3.6 Success and failure probabilities

A PLP has a single success outcome and many possible failure outcome cases. Each possible outcome has a certain probability that represented with fraction between 0 to 1, the sum of all outcomes must be 1. We represent probabilities with probabilities on edges coming out from a single node.

3.7 Side effects

Side effects are the changes made to variables or resources by PLP during its execution. The two most significant complications caused by side effects: 1. Temporal - when exactly the side effects influence the variables: it can be a single point in time, multiple points in time, single interval or multiple intervals. 2. Value - the value that the side effect assigns to a variable can be constant or change according to time and other variables. Although PLP does not represent temporal nuances of side effects, we can be confident that side effects have been applied before the end of the PLP. We will make sure to enforce the update of the side effect at the end of representing PTA_{PLP}.

3.8 *Observed variable*

PLP Observe is supposed to be able to obtain the value of a particular variable, as it appears in the environment. Although it is possible to model the environment of a robot, we decided not to over-complicate the system with an attempt to do so and to sustain a universal and probabilistic approach to the system. Therefore instead of just assigning a value from an environmental module, we use the domain of the specific *observed variable* to choose its probable value. Currently, we use only a uniform probability on variable's domain, but this feature can be expanded in the future.

3.9 Update attribute extension

We will be using functions embedded inside the update attribute of an edge. A Function is basically a series of assignments to variables and *clocks*, with an extension of conditional " if " statement. Conditional " if " statement evaluates a certain $if - condition$. If it is true, only the $true - part$ is evaluated, if the condition is false, only the $false - part$ is evaluated. " if " statement:

$$if(if - condition)\{true - part\}else\{false - part\}$$

A simple implementation of " if " statement, is to use two different edges with complementary parts of the " if " statement: 1. Edge with $if - condition$ as a *guard* attribute, and $true - part$ as a content of an update attribute. 2. Edge with $NOT(if - condition)$ as a *guard* attribute, and $false - part$ as a content of an update attribute.
For example, we create function check() that equals to: " $if(1 == a)\{b \leftarrow 2, c \leftarrow 1\}else\{b \leftarrow 3, c \leftarrow 3\}$ ". If a equals 1 then $true - part$ " $b \leftarrow 2, c \leftarrow 1$ " is executed, otherwise the $false - part$ " $b \leftarrow 3, c \leftarrow 3$ " is executed. We can write:

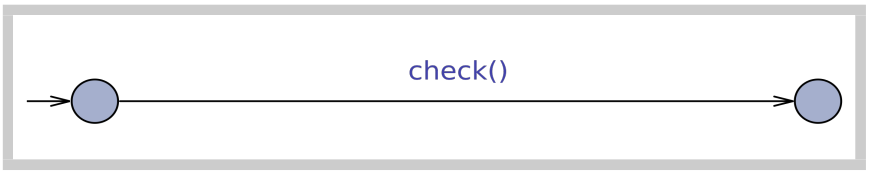


FIGURE 3.1

Implementation:

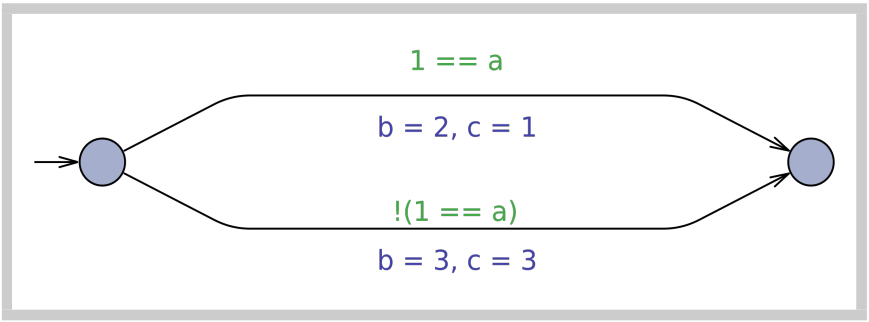


FIGURE 3.2

Nested " if " statements can be implemented by attaching the nested conditions with logical AND into *guard* attribute, and filling the update by response assignment of the most inner " if " (with an addition of common assignments).
The pseudocode of the functions we are using can be seen in Appendix A1.

3.10 *Concurrent condition*

Models can influence each other while running; this concept is captured in PLPs by *concurrent condition*, that can be accessed by two or more PLPs at the same time. Although we do not actually execute the code characterized by PLPs, our verification framework has to address the constraints introduced by *concurrent conditions*, in order to verify systems with more than just one parallel execution of PLP. If certain PLP can run only while some *concurrent condition* is satisfied, we need to verify that it is indeed satisfied. PTA_{PLP} simulate the notion of execution time, and we make sure that *concurrent conditions* of PLP are satisfied while PLP's execution is verified. To implement *concurrent condition* in PTA_{PLP}, we want to prevent race conditions and

incoherency of variables. This will be achieved by the *concurrent module* in 3.12.

3.11 *Termination conditions*

As we established regarding *concurrent condition*, our system should be able to handle many coexisting PLPs, which may interact with *termination conditions* of each other. *Termination condition* can cause PLP to finish the execution successfully or stop and enter one of the failure termination states. *Termination condition* may involve many variables, which if changed, any one of them can satisfy the *termination condition*. The *termination condition* was a particularly challenging aspect of PTA_{PLP} design because any delay between the change of variable's value by one PTA_{PLP} and reception of it by other PTA_{PLP} would mostly represent the limitations of the modeling environment and will cause ever-growing divergence between model and the real system, with an increase of the delay. That is why we had to eliminate the delay between variable's value change, and PTA_{PLP} acknowledgment of the change. We could not just use regular variables to achieve immediate communication between PTAs because any PTA that waits in one of its states is not obligated to respond imminently when *enabling condition* is satisfied, which will produce the undesired delay. We had to enforce PTA to notice any variable's change immediately. The obvious choice to achieve the desired effect is to use PTA channels that enable immediate and compulsory response between two or more PTAs. In our system design, we create a unique channel for each variable, which is used by the writing PTA, to signal others that variable's value has been changed. All other PTAs with *termination conditions* involving this variable, need to have a single edge waiting for a signal on a corresponding channel, with *termination condition* as their *enabling condition*. Therefore if one PTA writes a new value to a variable, it also has to signal on a correlated channel, and all other PTAs that waits on states with *termination conditions*, would receive the signal immediately and proceed to check *termination condition*. For complex *termination condition*, every included variable has to have a single edge with a correlated channel for this variable. For each variable change, the corresponding edge will check if the *termination condition* is true. If the *termination condition* is satisfied at some point, we may look on the last chronological change to some variable that is satisfying the whole condition. There has to be an edge with a dedicated channel for this variable, change of this last variable would be accompanied by a signal on a channel that will force an immediate check of the *termination condition* and transition to an appropriate state, without any delay.

For example, for variables: $a = true$ and $b = true$.
PTA_A is changing the value of *a* to *false* and signaling about the change on *a_chan* channel:

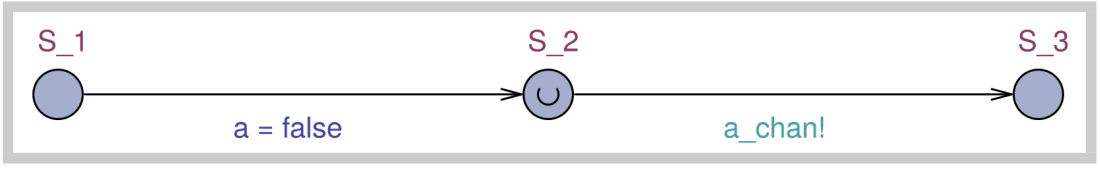


FIGURE 3.3

PTA_B waits in *S_1* state for *termination condition "not(a) and b"*:

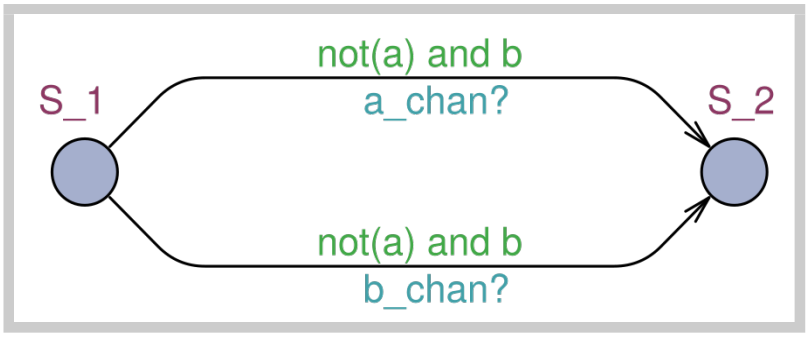


FIGURE 3.4

PTA_C waits in *S_1* state for *termination condition "not(a) and not(b)"*:

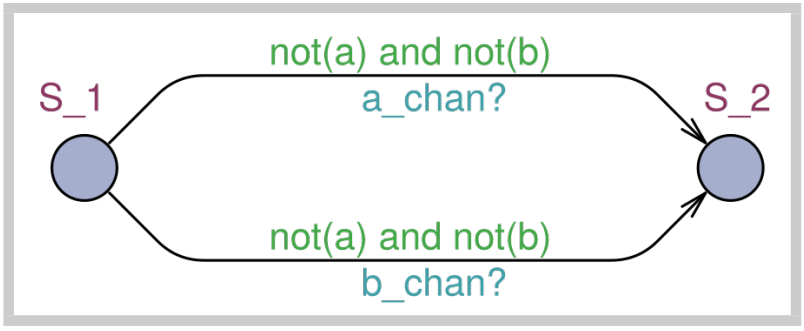


FIGURE 3.5

In this example when PTA_A transitioning from *S_2* to *S_3*, the variable *a* is already *false*, a signal to PTA_B and PTA_C passed on the channel *a_chan*. PTA_B *termination condition* is satisfied, and it can transition from *S_1* to *S_2*. PTA_C *termination condition* is not satisfied by the change; therefore PTA_C remains in the *S_1* state.

In our system any variable can be used as part of some *termination condition*; therefore we need to enforce the signaling mechanism that we described here, upon every write to any variable. Due to parallel execution of $PTAs_{PLP}$ in the system, we need to deal with race conditions and incoherency of data. We will address these problems with the *concurrent module* in 3.12.

3.12 *Concurrent module*

To handle a system with numerous PLPs interacting with each other through shared memory, we need to make sure to enforce restrictions on the parallel nature of our system. We need to make sure to prevent race conditions and incoherency of *concurrent conditions*, *termination conditions*, goals, preconditions, *observed variables*, resources, side effects, and generally any shared memory between $PTAs_{PLP}$. For this purpose, we dedicated a special memory in the system, and we enforce a mutually exclusive access to it between the $PTAs_{PLP}$. We refer to this memory as a *concurrent memory*. To the PTA that enforces mutually exclusive access to the *concurrent memory*, we call it the *concurrent module*.

The *concurrent module* provides the capability for PTA_{PLP} to check that values of certain variable remain inside (or outside) certain range even when accessed by other $PTAs_{PLP}$. The *concurrent module* combines the following components: 1. *Concurrent memory* - an array of variables and array of variable tracking requests to make sure they stay in a certain range. 2. *Concurrent module* PTA - PTA that is enforcing mutually exclusive access to data. 3 .*Concurrent module* API - functions (pseudocode in Appendix A1) and channels used to communicate with *concurrent module* PTA, to read/write data and add/remove tracking requests for variables.

We define a unique ID for each PTA_{PLP}, to make easy communication between $PTAs_{PLP}$ and *concurrent module* PTA. Variables and variable tracking requests have a unique ID used by $PTAs_{PLP}$ and *concurrent module* PTA.

The basic idea of the *concurrent module* is based on principles of multi-threaded mutual exclusion. Each PTA_{PLP} can be in a specific code section at any given time: 1. Entry section - waiting to enter the critical section. 2. Critical section - can access *concurrent memory*. 3. Exit section - letting others enter the Critical Section sometime later. 4. Remainder section - unrelated to other sections.

3.12.1 Access *concurrent module*

To gain access for the sake of reading/writing/requesting actions on the *concurrent memory* in a mutually exclusive manner, every PTA_{PLP} can use the protocol defined below that is composed of three required parts: 1. Entry section. 2. Critical section. 3. Exit section.
Here is an example of access to *concurrent module* by PTA_{PLP}:

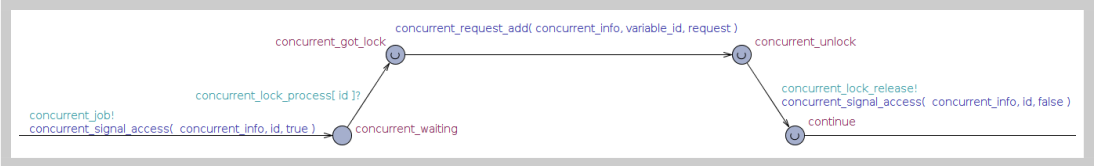


FIGURE 3.6

In this example, PTA_{PLP} is adding variable tracking request for variable *variable_id*.
These are the mutual exclusion sections from PTA_{PLP} point of view:

- Entry section - PTA_{PLP} that wishes to enter the critical section, must have an edge with the properties: 1. Update attribute is containing: *concurrent_signal_access* - a function that updates *is_process_would_like_to_access* variable for current PLP, and letting *concurrent module* PTA know that PTA_{PLP} wants to access the *concurrent memory*. 2. *Synchronization* attribute is containing: *concurrent_job!* - synchronization channel that allows *concurrent module* PTA advance from waiting to choosing a single PTA_{PLP} to enter the critical section. After passing the first edge, PTA_{PLP} is waiting on *concurrent_lock_process[id]* channel, for *concurrent module* PTA to let this PTA_{PLP} an access to the critical section of the *concurrent module*.
- Critical section - contains a single *urgent* state with an outgoing edge that including all the access to *concurrent memory*. The update attribute can contain the following functions:

- *concurrent_read* - read variable *concurrent_info.concurrent_data[variable_id].value* value . It can be accessed by *guard* attribute without synchronization.
- *concurrent_write* - write variable value. This is the only function that has to be followed by an edge with a signal on *concurrent_notify[variable_id]* channel, to inform other $PTAs_{PLP}$ that are waiting for this variable to change.
- *concurrent_request_add* - add a request to track a value of a certain variable. There are two types of possible requests: 1. Make sure the value is staying inside a certain range. 2. Make sure the value is staying outside a certain range.
- *concurrent_request_remove_request* - remove a request to track certain variable value, and return whether the value of the variable was as expected, according to the type of the request, since the *concurrent_request_add* for this variable.

- Exit Section - contains a single *urgent* state with an outgoing edge that calls to the *concurrent_signal_access* function, to signal that PTA_{PLP} is no longer need to access *concurrent memory*. The same edge also synchronized with the *concurrent module* on *concurrent_lock_release!* channel and lets the *concurrent module* PTA continue to run.

3.12.2 *Concurrent module* PTA

We present here the *concurrent module* PTA for a system with n $PTAs_{PLP}$, with IDs: 0,1,...,n-1. In the real system, the number of $PTAs_{PLP}$ would vary according to the amount of PLPs. Places with "// …" are placeholders for parts that can be easily filled if n is defined.
Concurrent module PTA:

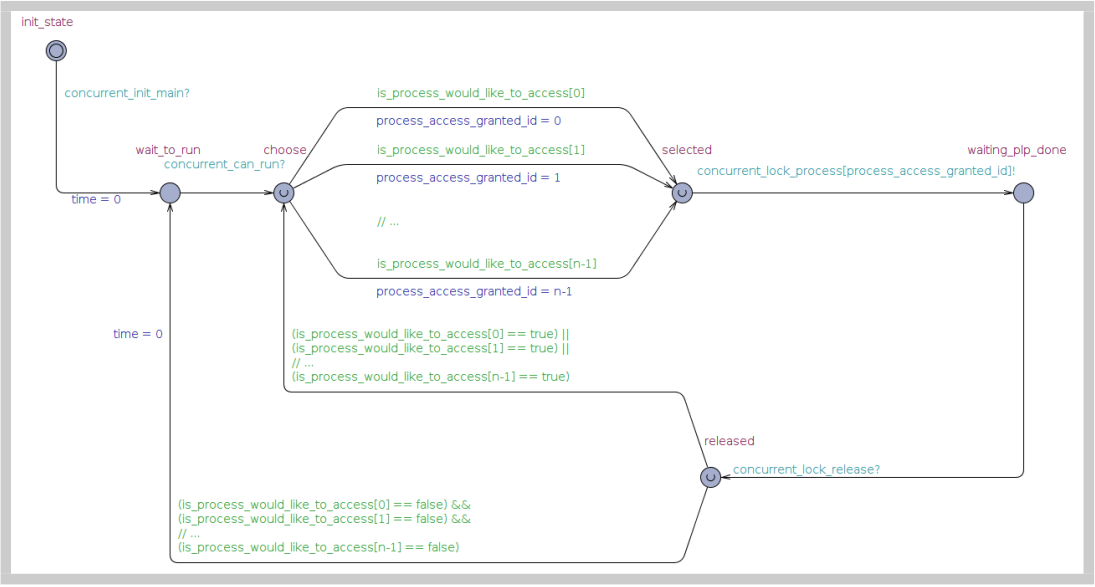


FIGURE 3.7

The transition from *init_state* to *wait_job* will occur after initialization of *concurrent_info* that contain all the variables, requests and supporting data.

The transition from *wait_job* to *choose* state will occur when any PTA_{PLP} will execute its entry section by the edge with *concurrent_signal_access* function and signaling on *concurrent_job* channel .

In the transition from *choose* to *selected* state, the *concurrent module* PTA selects a single PTA_{PLP} that would be granted access to the critical section in the next step. The *concurrent_lock_process* will let the PTA_{PLP} enter the critical section, and *concurrent module* PTA will wait until the PTA_{PLP} is done, which does not require any time to pass. Then PTA_{PLP} will signal on *concurrent_lock_release* that it is done. From the *released* state the *concurrent module* PTA can move to *wait_job* or *choose* states. Although the guards contain multiple variables, inconsistency and race conditions are prevented because edges executed as atomic units (as defined by model transition semantics). So there is a particular transition dedicated for this edge that does not interleave with other edges, and all the variables are checked by the guard of an outgoing edge from *released* state. If there is any PTA_{PLP} that want to enter a critical section, the *concurrent module* PTA will pass to *choose* state, and if there are no $PTAs_{PLP}$ that want to enter the critical section, *concurrent module* PTA will go to *wait_job* state.

Let us check that the presented algorithm satisfies the conditions expected from a decent solution for critical section problem:

1) Mutual exclusion.
   Claim: Two or more PTA_{PLP} cannot be in a critical section at the same time.
   Proof: We should show that when *concurrent module* PTA is located between *selected* and *released* states, only single PTA_{PLP} can access the API defined as the critical section.
   a) For two $PTAs_{PLP}$ to be at the same time at the critical section, they both need to receive signals on *concurrent_lock_process* for 2 different IDs.
   b) But only one PTA_{PLP} is chosen by *concurrent module* PTA between *choose* and *selected* states by a nondeterministic choice of a single edge, of PTA_{PLP} that want to access critical section (*is_process_would_like_to_access[id] == true*), for each pass on that edge.
   c) Any PTA_{PLP} that entered the critical section will have to pass the edge between waiting and *released* state, to let another PTA_{PLP} enter the critical section, but that is mean that PTA_{PLP} is signaling on the *concurrent_lock_relese* channel that is part of its exit section. Therefore Mutual Exclusion condition is holding.

2) Progress.
   Claim: If there are $PTAs_{PLP}$ that want to enter a critical section, eventually one of them will be able to enter the critical section.
   Proof: We need to show that if there are $PTAs_{PLP}$ that want to access the critical section, at least one of them can enter and exit the critical section.
   a) Without any $PTAs_{PLP}$, *concurrent module* PTA is waiting on *wait_job* state.
   b) If there is a single PTA_{PLP} that want to enter the critical section, it will have an edge with update *concurrent_signal_access* function that will be executed first and then will signal on *concurrent_job* channel, that will force transition from *wait_job* to *choose* state in *concurrent module* PTA.
      i) The transition between *choose* and *selected* states, the *concurrent module* PTA can make a nondeterministic choice of a single edge, only for PTA_{PLP} that want to access critical section (*is_process_would_like_to_access[id] == true*), this will assign a value to the *process_access_granted_id* variable.

ii) The transition between *selected* and *waiting* state will pass a signal on *concurrent_lock_process[process_access_granted_id]* channel to single PTA_{PLP} with *process_access_granted_id* ID.

iii) The chosen PTA_{PLP} will enter the critical section and access the *concurrent module* API, but it can not influence the *concurrent module* PTA unless it is exiting the critical section by signaling on the *concurrent_lock_relese* channel. But from this point, PTA_{PLP} is no longer in the critical section.

iv) From the *released* state, the *concurrent module* PTA will move back to *wait_job* state.

c) If there is more than one PTA_{PLP} that want to enter the critical section, the first one will force the transition from *wait_job* to *choose* state. Then as we seen for a single case, one PTA_{PLP} will enter and eventually exit the critical section, and *concurrent module* PTA will be at *released* state. Let us look on other PTA_{PLP} that wish to enter the critical section; it must execute the edge in its entry section with *concurrent_signal_access* function and *concurrent_job!* synchronization. But this edge can not interleave with the outgoing edge from the *released* state, so one edge will be executed before the other: 1. If the edge in entry section was executed first, it updated *is_process_would_like_to_access*, then when an outgoing edge from the *released* state will check the guards, only the guard of an edge going to *choose* state will be true, and the *concurrent module* PTA will transition to *choose* state. 2. If the edge from the *released* state was first, and there is another PTA_{PLP} that updated *is_process_would_like_to_access*, then the *concurrent module* PTA will move to *choose* state; otherwise there are no other $PTAs_{PLP}$ that want to enter the critical section yet, *concurrent module* PTA will go to *wait_job* state. Then will be executed the edge from entry section of the PTA_{PLP}, that want to enter the critical section, it will update *is_process_would_like_to_access* signal on *concurrent_job* channel, so *concurrent module* PTA will go to *choose* state. The transition from *choose* state will happen as we already explained. Therefore the progress condition is holding.

3) Starvation freedom - If $PTAs_{PLP}$ can exit critical section and request again to enter the critical section without any time passing, starvation freedom does not guaranteed, because *concurrent module* PTA can always go through *choose* state that will keep choosing the same $PTAs_{PLP}$, therefore starving other $PTAs_{PLP}$. But if $PTAs_{PLP}$ must take some time in remainder section, which is probably the more realistic scenario, starvation freedom would be guaranteed, because the cycle *choose* to *released* states containing only *urgent* states (*waiting* state occurs between *urgent* states in PTA_{PLP}), is taking no time at all. So the set of $PTAs_{PLP}$ waiting to enter the critical section will decrease with each PTA_{PLP} entering the critical section, due to guaranteed progress they will exit eventually and wait before entering.

3.12.3 *Concurrent module* applications

There are eight possible applications of the *concurrent module*: 1. Collaborative modules - PTA_{PLP} need to run at the same time with a set of other $PTAs_{PLP}$. 2. Contender modules - $PTAs_{PLP}$

cannot run at the same time with a set of other $PTAs_{PLP}$. 3. *Concurrent condition hold* - PTA_{PLP} *Maintain* holding variables at certain values. 4. *Concurrent condition check* - any PTA_{PLP} can check the value of a variable that some PTA_{PLP} *Maintain* is holding, and decide whether its value is acceptable or not. 5. *Goal condition* update - PTA_{PLP} *Achieve* and PTA_{PLP} *Detect* completed successfully and therefore need to update variables that appear in their *goal condition*. 6. *Observed variable update* - PTA_{PLP} *Observe* completed successfully and therefore needs to update the variable that was observed. 7. *Side effects update* - any PTA_{PLP} when completed successfully and therefore needs to update the variables in the side effects. 8. *Required resource check* - any PTA_{PLP} may need to check that there are enough resources for it to run, with exclusive access to resources.

We will split *concurrent memory* into two areas: 1. An area used to indicate which modules are currently running, with a dedicated variable representing the possible running state of the PTA_{PLP}, value 0 for not running and value 1 for running. 2. An area used by PTA_{PLP} *Maintain* and other $PTAs_{PLP}$ that check values.

The general structure of implementation for all cases consists of the following sequential components: 1. Preliminary *concurrent memory* access. 2.PTA_{PLP} time passage. 3. Concluding *concurrent memory* access. 4. Assessing results. PTA_{PLP} time passage is an independent component determined by PLP run time distribution, so we will ignore it for now in the context of explaining *concurrent module*. We get a coarse PTA structure:

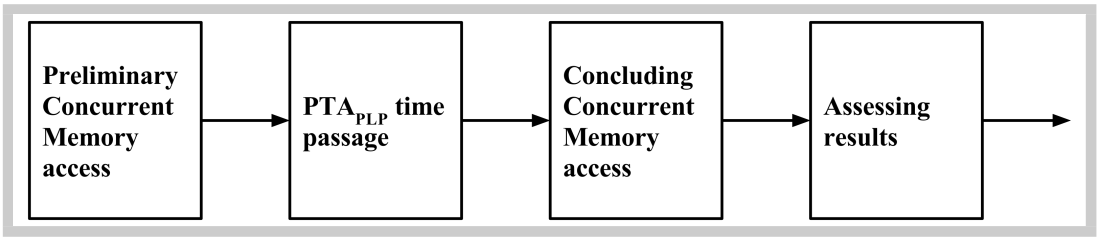


FIGURE 3.8

We will describe the implementation of each application, from the perspective of current PTA_{PLP} and perspective of other concurrent PTA_{PLP} that is running concurrently to current PTA_{PLP}:

3.12.3.1 Collaborative modules

Current PTA_{PLP} can work only while certain concurrent PTA_{PLP} is running: (value 1 means that other PTA_{PLP} is running, and value 0 means that other PTA_{PLP} is not running)

1) Current PLP:
    a) Preliminary access: call for *concurrent_request_add* function with a variable for concurrent PTA_{PLP}, and acceptable value 1.
    b) Concluding access: call for *concurrent_request_remove_request* function for original request, store result to *result* variable.
    c) Assessing results: if *result == false* then go to the failure state, otherwise continue running.

2) Concurrent PTA_{PLP}:

   a) Preliminary access: call for *concurrent_write* function with a variable for concurrent PTA_{PLP}, with value 1.
   b) Concluding access: call for *concurrent_write* function with a variable for concurrent PTA_{PLP}, with value 0.
   c) Assessing results: always continue.

3.12.3.2 Contender modules

Current PTA_{PLP} can not work while certain concurrent PTA_{PLP} is running: (value 1 means that other PTA_{PLP} is running, and value 0 means that other PTA_{PLP} is not running)

1) Current PTA_{PLP}:

   a) Preliminary access: call for *concurrent_request_add* function with variable for concurrent PTA_{PLP}, and acceptable value 0.
   b) Concluding access: call for *concurrent_request_remove_request* function for original request, store result to *result* variable.
   c) Assessing results: if *result == false* then go to the failure state, otherwise continue running.

2) Concurrent PTA_{PLP}:

   a) Preliminary access: call for *concurrent_write* function with a variable for concurrent PTA_{PLP}, with value 1.
   b) Concluding access: call for *concurrent_write* function with a variable for concurrent PTA_{PLP}, with value 0.
   c) Assessing results: always continue.

3.12.3.3 *Concurrent condition hold*

PTA_{PLP} Maintain holding variables at certain values.

1) There is a subtle point involving *PTA_{PLP} Maintain*, that 2 or more different $PTAs_{PLP}$ can maintain contradicting values of the same variable. Depending on the particular system design such contradicting *$PTAs_{PLP}$ Maintain* could be unacceptable and imply that there are problems with the system, while at other systems contradicting values could be totally acceptable and even expected. In more peculiar cases we can expect mixed behavior, depending on the particular variable. To deal with this problem, we decided to use by default the more cautious approach, so any contradicting *$PTAs_{PLP}$ Maintain* is causing one of the involving PTA_{PLP} to enter failure case. But we also provide the ability for the user to specify in the configuration file exactly which variable should be verified for contradiction value and which variables can be ignored.
2) Verify Contradictions:

   a) Preliminary access:

      i) Call for *concurrent_write* function with variable ID and value as specified.

ii) Call for *concurrent_request_add* function with variable ID and acceptable as written in the previous line.

b) Concluding access: call for *concurrent_request_remove_request* function for original request, store result to *result* variable.

c) Assessing results: if *result == false* then go to the failure state, otherwise continue running.

3) Ignore Contradictions:

a) Preliminary access: call for *concurrent_write* function with variable ID and value as specified.

b) Concluding access: empty.

c) Assessing results: empty.

3.12.3.4 *Concurrent condition check*

Check that value maintained by some PTA_{PLP} *Maintain* is acceptable by this PTA_{PLP}.

1) Preliminary access: call for *concurrent_request_add* function with variable ID and acceptable values.
2) Concluding access: call for *concurrent_request_remove_request* function for original request, store result to *result* variable.
3) Assessing results: if *result == false* then go to the failure state, otherwise continue running.

3.12.3.5 *Goal condition* update

PTA_{PLP} *Achieve* and PTA_{PLP} *Detect* that completed successfully need to update variables that appear in their *goal condition*. *Goal condition* is converted into a series of assignments. For each assignment:

1) Preliminary access: empty.
2) Concluding access: call for *concurrent_write* function with variable ID from the assignment and value as specified in the assignment.
3) Assessing results: empty.

3.12.3.6 *Observed variable update*

PTA_{PLP} *Observe* that completed successfully needs to update the *observed variable* with a value from variable's domain, with uniform distribution.

1) Preliminary access: empty.
2) Concluding access: call for *concurrent_write* function with variable ID for the *observed variable* and value according to the distribution.
3) Assessing results: empty.

3.12.3.7 *Side effects update*

Any PTA_{PLP} when completed successfully needs to update the variables in the side effects.

1) Preliminary access: empty.
2) Concluding access: call for *concurrent_write* function with variable ID for the side effect variable and value according to the effect.
3) Assessing results: empty.

3.12.3.8 *Required resource check*

Any PTA_{PLP} may need check that there are enough resources for it to run, with an exclusive access to resource.

1) Preliminary access:
    a) Call for *concurrent_request_add* function with a variable ID for resource variable and minimum value needed for the resource.
    b) Call for *concurrent_request_add* function with a variable ID for resource variable and current value of the resource variable, to make sure nobody else is using it.
2) Concluding access: call for *concurrent_request_remove_request* function for both original requests, store result to *result_1* and *result_2* variable.
3) Assessing results: if *result_1 == false* or *result_2 == false* then go to the failure state, otherwise continue running.

Here we can see an example of *concurrent module* application for PTA_{PLP} *Maintain*, that is maintaining the value 5 and verifying it:

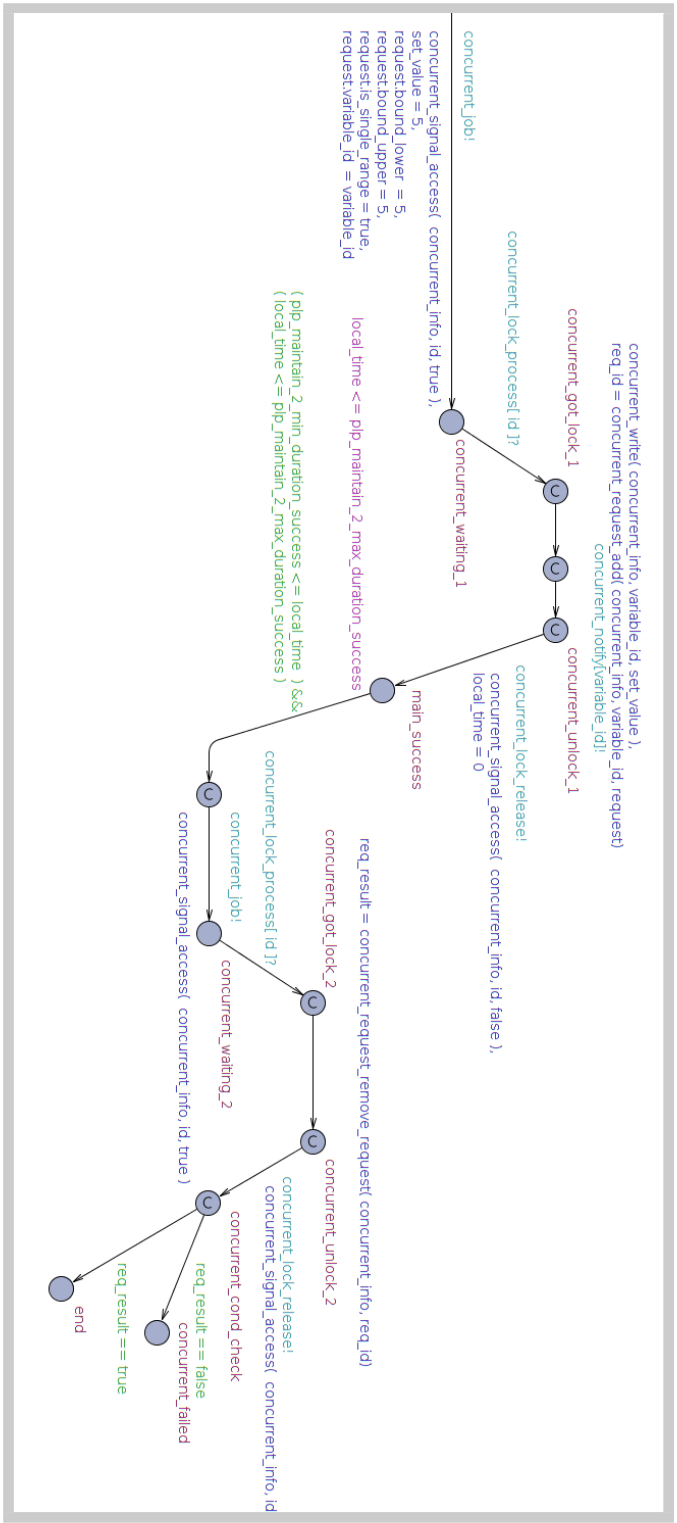

concurrent_write(concurrent_info, variable_id, set_value),
req_id = concurrent_request_add(concurrent_info, variable_id, request)
concurrent_notify(variable_id)!
concurrent_got_lock_1
concurrent_unlock_1
concurrent_lock_process[id]?
concurrent_lock_release!
concurrent_signal_access(concurrent_info, id, false),
local_time = 0
concurrent_job!
concurrent_waiting_1
concurrent_signal_access(concurrent_info, id, true),
set_value = 5,
request.bound_lower = 5,
request.bound_upper = 5,
request.is_single_range = true,
request.variable_id = variable_id
local_time <= plp_maintain_2_max_duration_success
main_success
(plp_maintain_2_min_duration_success <= local_time) &&
(local_time <= plp_maintain_2_max_duration_success)
req_result = concurrent_request_remove_request(concurrent_info, req_id)
concurrent_got_lock_2
concurrent_unlock_2
concurrent_lock_release!
concurrent_signal_access(concurrent_info, id
concurrent_lock_process[id]?
concurrent_job!
concurrent_waiting_2
concurrent_signal_access(concurrent_info, id, true)
concurrent_cond_check
req_result == false
concurrent_failed
req_result == true
end

FIGURE 3.9

CHAPTER 4

Formal Semantics for PLPs

In this section, we will continue with the premises from the previous section (see: 3) and use the established foundation, to follow specifications of each type of PLP, with the goal to define a formal semantics for it with a PTA model.

4.1 PTA_{PLP} *Achieve*

PLP Achieve describes modules that work to accomplish a certain goal. We use PTA's components that we have already described: conditions, time distributions, and probabilities, to create template PTA for PTA_{PLP} *Achieve*.
PTA_{PLP} *Achieve* template:

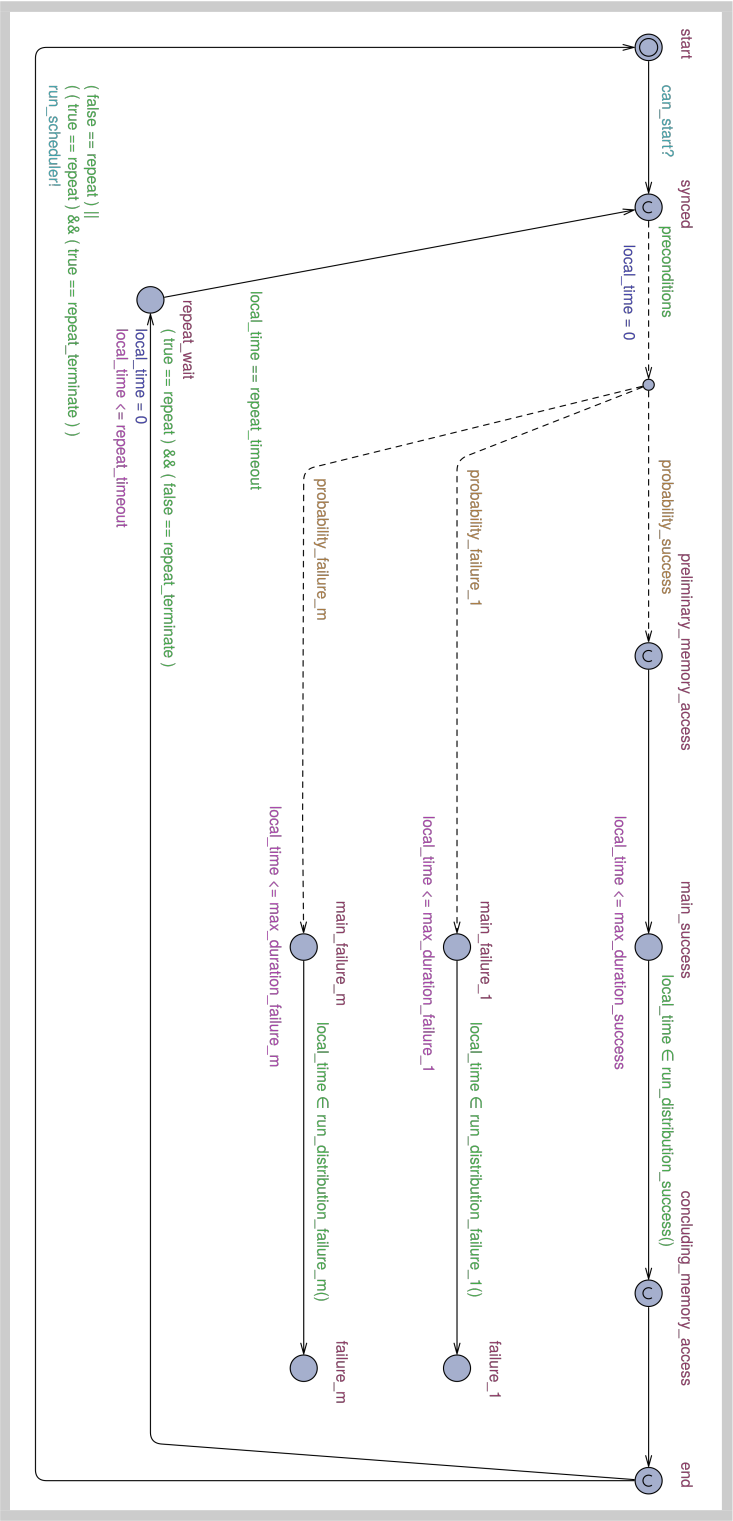


FIGURE 4.1

We will go through the elements of *PLP Achieve* (see: 2.1.1 and 2.1.2) and show how they are defined by PTA_{PLP} *Achieve*.

4.1.1 Scheduling mechanism

We assume that there is some other PTA that plays the role of the scheduler. For our PTA_{PLP} to start, the *scheduler* passes a signal on the *can_start* channel. The PTA_{PLP} is advancing from *start* to *synced* state when a signal from *scheduler* received, which allows PTA_{PLP} to start. When PTA_{PLP} completed, the transition from *end* to *start* state passes a signal on the *run_scheduler* channel, which signals *scheduler* to proceed.

4.1.2 Preconditions

In order to start working PLP needs all its precondition to be fulfilled, which we convert from PLP (see: 3.4) form into PTA conditions (see: 3.2). Preconditions are placed at *guard* attribute after *synced* state. If all preconditions fulfilled, PTA_{PLP} can work; otherwise, it will create a deadlock.

4.1.3 Success and failure modes, and probabilities

Success and failure modes represented by a unique state for each mode: *main_success*, *main_failure_1*, ... , *main_failure_m*. After *synced* state, there is a probabilistic choice of the possible outcome of the PTA_{PLP}: *probability_success*, *probability_failure_1*, …, *probability_failure_m*, corresponding to the probabilities of modes described in PLP (see: 3.6). There are *m* possible failure cases that each represented by its branch. PTA_{PLP} will choose a single edge to continue from the *synced* state, which will decide the outcome of PTA_{PLP}.

4.1.4 Run time distribution

The states *main_success*, *main_failure_1*, ... , *main_failure_m* are the states in which time can pass for this PTA_{PLP}, according to the time distributions specified in PLP for each success and failure mode. Run time distributions are *guard* attributes of the outgoing edges from these states: *local_time* $\in$ *run_distribution_success()*, *local_time* $\in$ *run_distribution_failure_1()*, ... , *local_time* $\in$ *run_distribution_failure_m()*.

4.1.5 *Goal condition*

Goal condition is extracted from PLP and converted (see: 3.2) to series of assignments as part of access to *concurrent memory* in *concluding_memory_access* (see: 3.5 and 3.12.3.5).

4.1.6 Resources requirements

PLP may require a certain amount of resources. For every resource needed by a PLP, there exists a designated variable to store the currently available amount of this resource and the

minimum required amount of it. We use *concurrent module* (see: 3.12) to make sure the amount is sufficient, and it is not being used by other PTA_{PLP} (see: 3.12.3.8).

4.1.7 *Concurrent conditions* and modules

Concurrent conditions and modules are implemented by *concurrent module* (see: 3.12). The accesses to the *concurrent memory* will be placed before and after the *main_success* state, in place of *preliminary_memory_access* and *concluding_memory_access* states. Possibly a failure state will be added to check the result from the concurrent request.

4.1.8 Side effects

As we already discussed (see: 3.7), the side effect would be accessed once, after *main_success* state, at *concluding_memory_access* state, as part of accesses to *concurrent memory* (see: 3.12.3.7).

4.1.9 Repeat extension

Repeat extension (see: 2.1.6) forces PLP loop until terminated. Repeat just makes the PTA_{PLP} run until the *termination condition* is true. It is implemented by adding the *repeat_wait* state between *end* and *synced* state. When in repeat mode and not terminated, PTA_{PLP} will wait in the *repeat_wait* state, as a way to represent *execution frequency*, and then it will continue to execute the PTA_{PLP} as before. Repeat *termination condition* is not immediate, so we represent it with a boolean *repeat_terminate*. If done repeating, PTA_{PLP} will let *scheduler* to run, by passing a signal on the *run_scheduler* channel.

4.2 PTA_{PLP} *Maintain*

PLP Maintain describes a module that makes sure that a certain condition is holding while running, it can be terminated spontaneously by run time distribution or by *termination condition*. The PTA_{PLP} *Maintain* will be similar to PTA_{PLP} *Achieve*, with changes and additions.
PTA_{PLP} *Maintain* template:

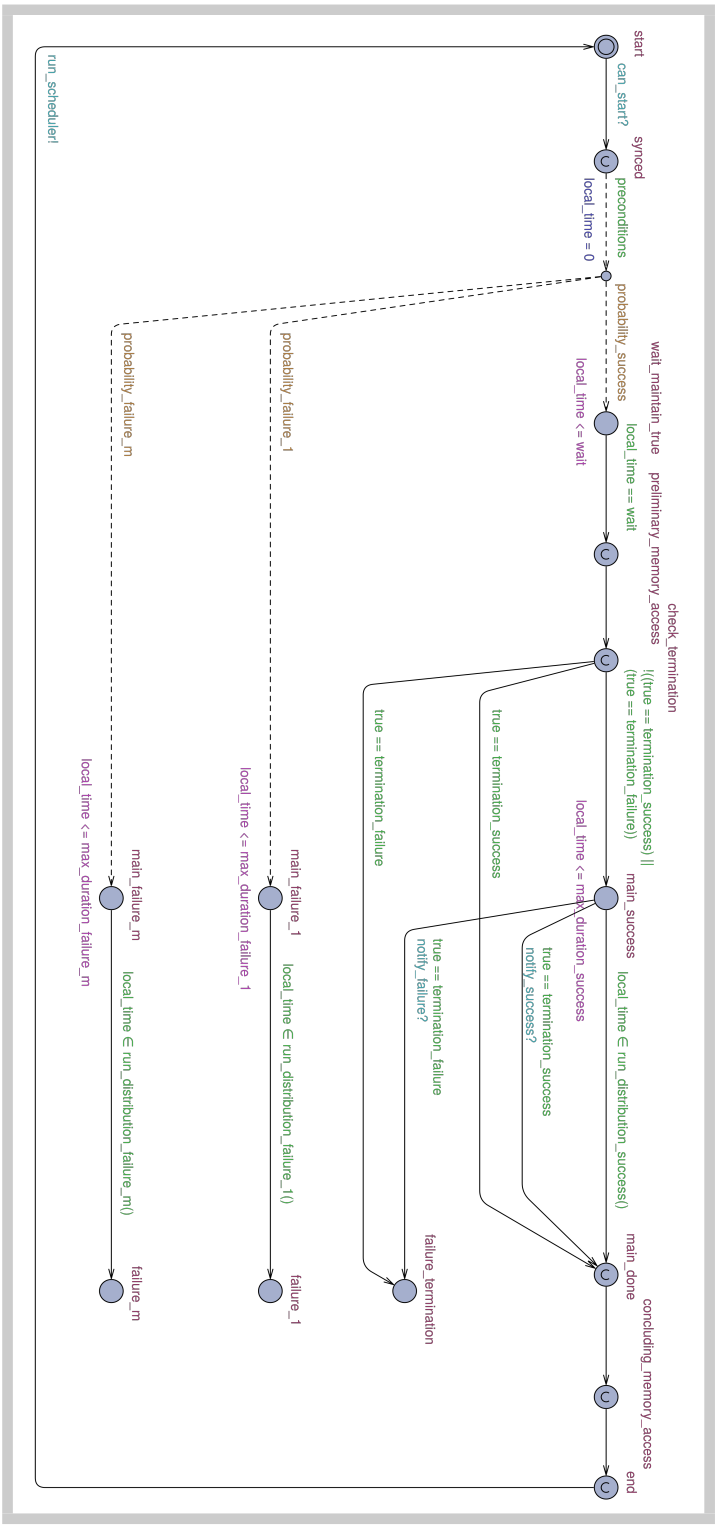


FIGURE 4.2

We will go through the elements of *PLP Maintain* (see: 2.1.1 and 2.1.3) and show how they

are defined by PTA_{PLP} *Maintain*.

4.2.1 Scheduling mechanism

Scheduling mechanism is defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.1).

4.2.2 Preconditions

Preconditions are defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.2).

4.2.3 Success and failure modes, and probabilities

Success and failure modes, and probabilities are defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.3).

4.2.4 Run time distribution

Run time distribution is defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.4).

4.2.5 Maintained condition

The maintained condition may be not true initially; we will be waiting in *wait_maintain_true* state *wait* until it becomes true. We will use *concurrent memory* (see: 3.12) to maintain condition, in a manner that we have already seen (see: 3.12.3.3 and 3.12.3.4). With *preliminary_memory_access* and *concluding_memory_access* states as placeholders for accesses to the *concurrent module*.

4.2.6 Resources requirements

Resources requirements are defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.6).

4.2.7 *Concurrent conditions* and modules

Concurrent conditions and modules are defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.7).

4.2.8 Side effects

Side effects are defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.8).

4.2.9 *Termination conditions*

PLP can be terminated immediately by failure or success *termination conditions*; we enforce the same termination restrictions on PTA_{PLP} (see: 3.11). *Termination conditions* are checked first in the *guard* attribute of the edge between *check_termination* and *main_success* state; before time can pass in the *main_success* state. If the *termination_success* condition is true, then we move to the *main_done* state. If the *termination_failure* condition is true, then we move to the *failure_termination* state. There can be more than one failure condition which will lead to its unique failure *failure_termination* state. If neither of the *termination conditions* satisfied before *main_success* state, we will go to *main_success* state. If *termination condition* satisfied while in *main_success* state, the other PTA_{PLP} that is writing the change to one of the termination variables, will also pass a signal on a dedicated channel, that with combination of enabled guard (that containing *termination condition*) will transfer the PTA_{PLP} *Maintain* into *main_done* or *failure_termination* state (according to termination case). More details in 3.11.

4.3 PTA_{PLP} *Observe*

PLP Observe senses a value of a certain variable from the real world. We do not support emulation of environment and state of the world, so we have to produce possible values of the environmental variables ourselves. It is being done by assignment of possible values to the variable according to the possible domain of the variable, with uniform distribution.
PTA_{PLP} *Observe* template:

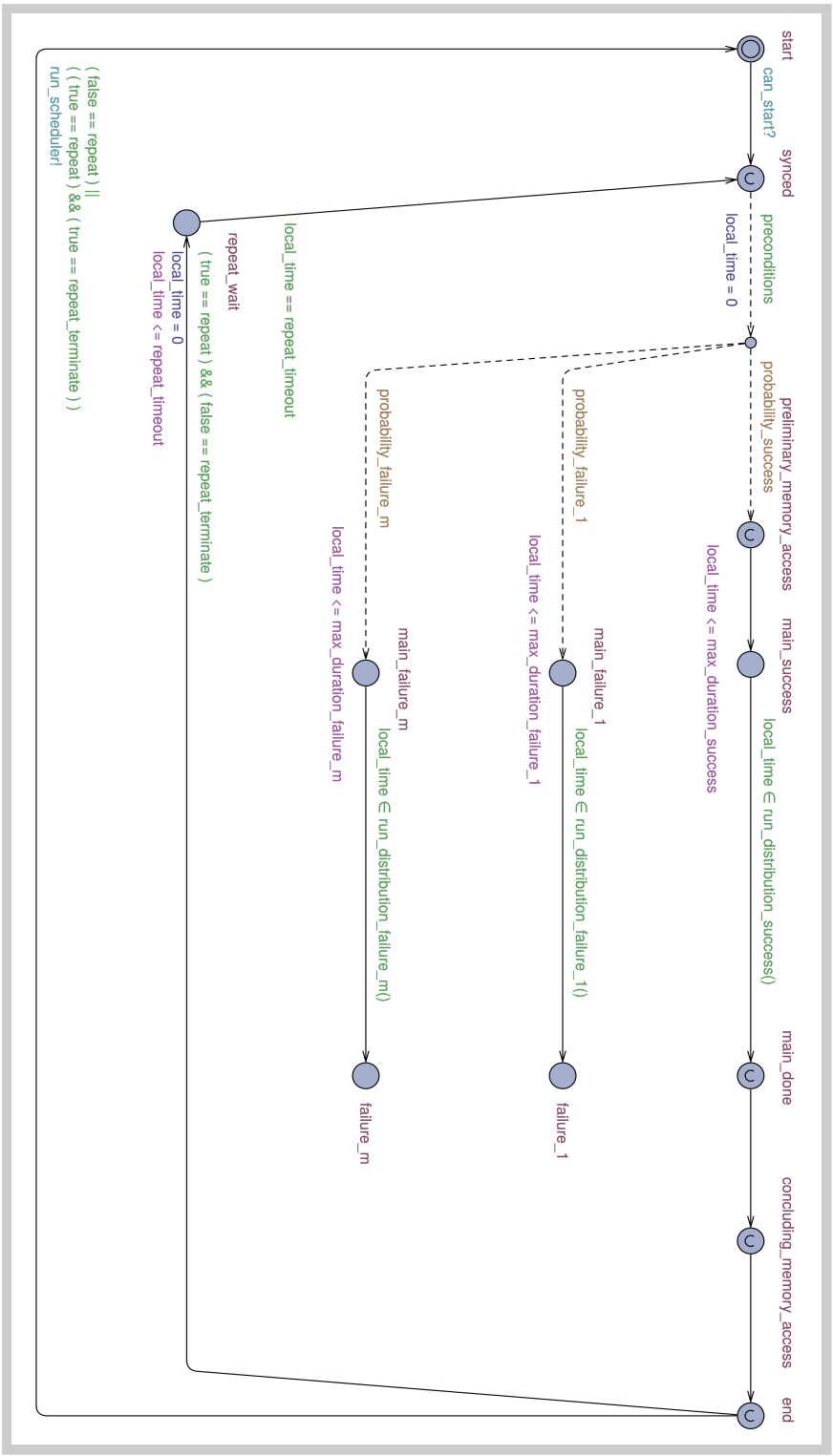


FIGURE 4.3

We will go through the elements of *PLP Observe* (see: 2.1.1 and 2.1.4) and show how they are defined by PTA_{PLP} *Observe*.

4.3.1 Scheduling mechanism

Scheduling mechanism is defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.1).

4.3.2 Preconditions

Preconditions are defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.2).

4.3.3 Success and failure modes, and probabilities

Success and failure modes, and probabilities are defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.3).

4.3.4 Run time distribution

Run time distribution is defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.4).

4.3.5 *Observed variable*

PLP Observe acquires a value of a variable from an environment, while PTA_{PLP} *Observe* chooses a probabilistically possible value for the *observed variable* (see: 3.8), with a uniform distribution upon *observed variable*'s domain. We are using *concurrent module* to update the value of the *observed variable* (see: 3.12.3.6).

4.3.6 Resources requirements

Resources requirements are defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.6).

4.3.7 *Concurrent conditions* and modules

Concurrent conditions and modules are defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.7).

4.3.8 Side effects

Side effects are defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.8).

4.3.9 Repeat extension

Repeat extension is defined identically to the case of PTA_{PLP} *Achieve* (see: 4.1.9).

4.4 PTA_{PLP} *Detect*

PLP Detect is waiting until a certain condition is fulfilled.
PTA_{PLP} *Detect* template:

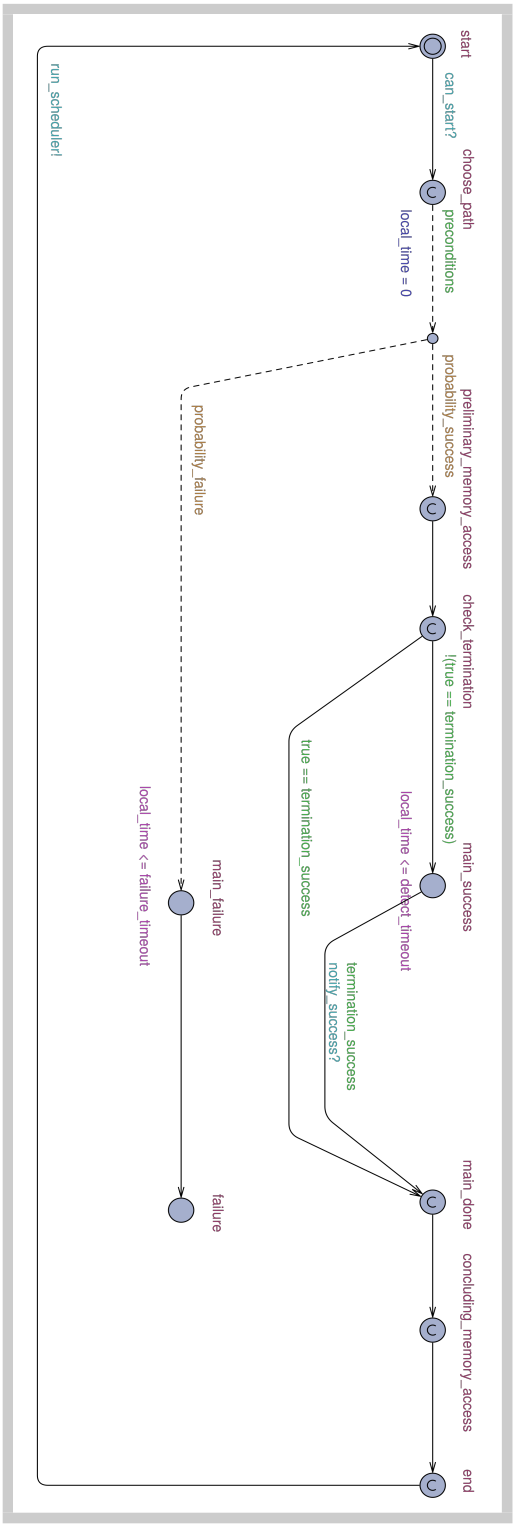


FIGURE 4.4

We will go through the elements of *PLP Detect* (see: 2.1.1 and 2.1.5) and show how they are defined by PTA_{PLP} *Detect*.

4.4.1 Scheduling mechanism

Scheduling mechanism is defined identically to the case of *PTA$_{PLP}$ Achieve* (see: 4.1.1).

4.4.2 Preconditions

Preconditions are defined identically to the case of *PTA$_{PLP}$ Achieve* (see: 4.1.2).

4.4.3 Success and failure modes, and probabilities

Success and failure modes represented by a single state for success *main_success* and a single state for failure *main_failure*. After *synced* state, there is a probabilistic choice of the possible outcome of the PTA$_{PLP}$: *probability_success* and *probability_failure*, corresponding to the probabilities of modes described in PLP (see: 3.6). PTA$_{PLP}$ will choose a single edge to continue from the *synced* state, which will decide the outcome of PTA$_{PLP}$.

4.4.4 Run time distribution

The time can pass in *main_success* and *main_failure* states, but *Detect PLP* defined not to support run time distributions. It simply stops when detection goal is satisfied. Similarly, in PTA$_{PLP}$ there are no run time distributions.

4.4.5 Resources requirements

Resources requirements are defined identically to the case of *PTA$_{PLP}$ Achieve* (see: 4.1.6).

4.4.6 *Concurrent conditions* and modules

Concurrent conditions and modules are defined identically to the case of *PTA$_{PLP}$ Achieve* (see: 4.1.7).

4.4.7 Side effects

Side effects are defined identically to the case of *PTA$_{PLP}$ Achieve* (see: 4.1.8).

4.4.8 Detection goal and *termination conditions*

PLP Detect can be terminated immediately by a detection goal that we refer to as a success *termination condition*; we enforce the same termination restriction on PTA_{PLP} (see: 3.11). The *termination condition* is checked first in the *guard* attribute of the edge between *check_termination* and *main_success* state; before time can pass in the *main_success* state. If the *termination_success* condition is true, then we move to the *main_done* state. If the *termination condition* is not satisfied before the *main_success* state, we will go to the *main_success* state. If the *termination condition* is satisfied while in the *main_success* state, the other PTA_{PLP} that is writing the change to one of the termination variables, will also pass a signal on a dedicated channel, that with a combination of the enabled guard (that containing *termination condition*) will transfer the PTA_{PLP} *Detect* into the *main_done* state. More details in 3.11.

CHAPTER 5

Control graph

We use the *control graph* as a way to describe a plan of execution of robotic models specified by PLPs, enhanced with probabilities, conditional branching, parallel execution, etc. We define a *control graph* as a directed graph that describes an execution flow of a control system. *Control graph* execution always starts at root node with a single execution path, that advances according to conditions and probabilities on edges. When the *concurrent node* is executed, it will split the execution path to multiple paths with outgoing edges.

Each node type comes in two variations:

1. Executed only when ALL of its incoming edges are executed.
2. Executed when ANY of its incoming edges are executed.

Outgoing edges can update the values of variables.

5.1 Sequence of $PTAs_{PLP}$

Nodes that execute a sequence of $PTAs_{PLP}$.

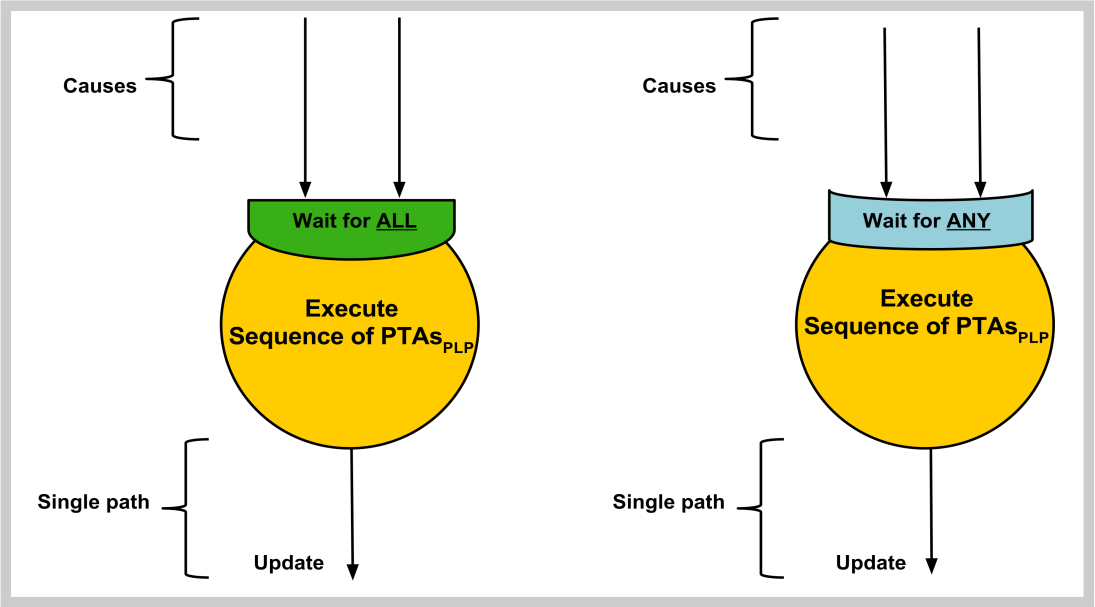


FIGURE 5.1

5.2 *Probabilistic nodes*

The *probabilistic node* would choose a single edge to proceed with according to the probability of the edge. The node also supports an optional delay.

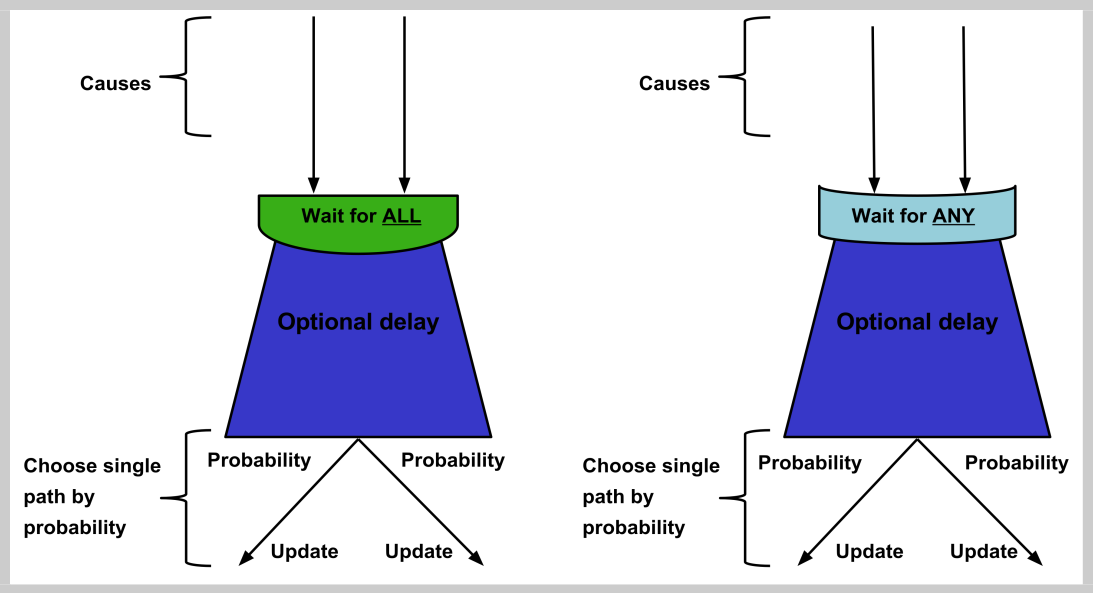


FIGURE 5.2

5.3 *Conditional nodes*

The *conditional node* would choose a single edge to proceed with, according to the condition of the edge that must be satisfied. If more than one condition is true, the path would be chosen non-deterministically between all the satisfied conditions. The node also supports an optional delay.

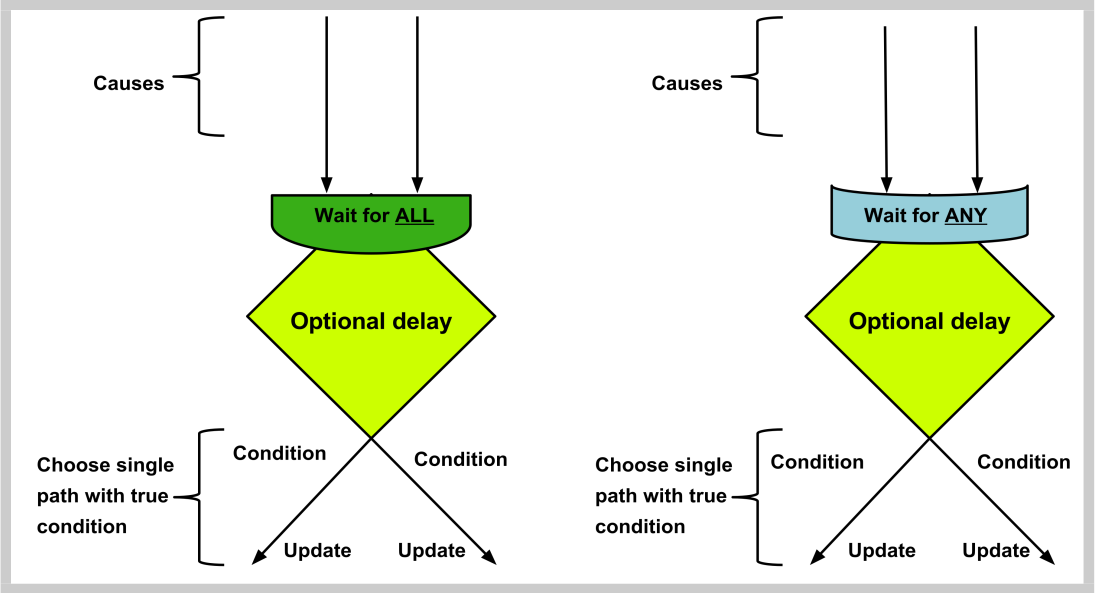


FIGURE 5.3

5.4 *Concurrent nodes*

The *concurrent node* would execute all the edges concurrently. The node also supports an optional delay.

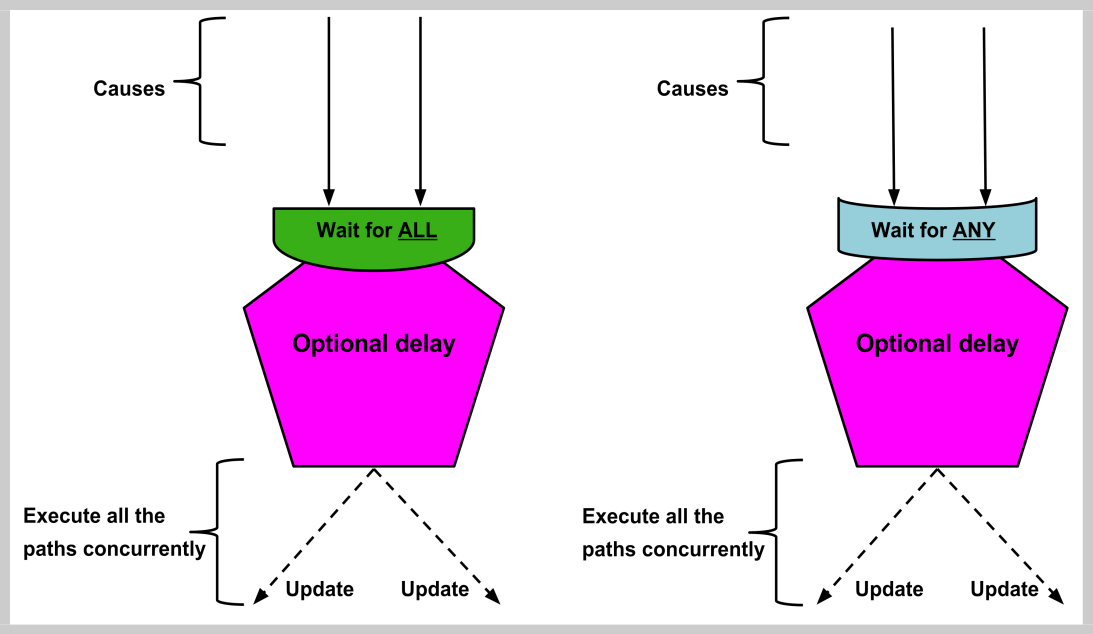


FIGURE 5.4

CHAPTER 6

Implementation with UPPAAL

6.1 UPPAAL

UPPAAL is a software package developed jointly by Uppsala and Aalborg Universities, for modeling, validation, and verification of real-time systems modeled as networks of timed automata, extended with data types [19]. UPPAAL allows us to create a set of PTAs that run in parallel, share variables and signal each other.
The following paragraphs constitute the essential introduction to UPPAAL needed for following chapters.

6.1.1 Data types

UPPAAL support declaration of variables of the following types that can be private a variable of a certain PTA or global variables shared between PTAs.

6.1.1.1 Boolean

True or false values.
Example:

```
bool b = true;
```

6.1.1.2 *Integer*

Integer numbers in the range [-32768, 32767].
Example:

```
int i = 0;
```

6.1.1.3 Double

Double precision floating point numbers. They are ignored by model checking parts, so we will not be using them much.
Example:

```
double d = 0.0;
```

6.1.1.4 Arrays

Similar to arrays in many programming languages, bundle together a group of variables of the same type, that we can refer to by the index in the array.
Example:

```
int  int_arr[ 3 ] = { 1, 1, 2 };
bool bool_arr[ 4 ] = { true, true, false, true };
```

6.1.1.5 Structure

Similar to the struct in C programming language, make possible to create more complex data structures.
Example:

```
typedef struct {
        bool b;
        int i;
} my_struct;
my_struct v = { true, 1 };
```

6.1.2 PTA

PTA consists of nodes (locations) and edges (discrete transitions). The node includes the properties: 1. Name of the node. 2. Is it the initial node of the PTA. 3. Is it an *urgent* node that time cannot pass in it. 4. An *invariant condition* - must remain satisfied for PTA to stay in the node. Edge includes the properties: 1. Guard is the enabling transition to pass the edge. 2. Synchronization - sending or receiving synchronization on some channel. 3. Update - *clock* reset or assignments to other variables.

6.1.3 Synchronization mechanism

Sometimes, we need PTAs to signal each other instantly, so two or more PTAs can be sure that they are passing a certain edge in the same exact moment.
Let us look at an illustration of two PTAs: PTA_A and PTA_B, we assume that PTA_A wants to signal PTA_B when it is passing the edge E in PTA_A. The synchronization mechanism uses edge's *attribute* dedicated for synchronization with a global variables *sync_channel* of type chan (channel). PTA_B would wait in a certain location, that has an outgoing edge F. When PTA_A would go through edge E, it will pass a signal on a channel *sync_channel*. PTA_B will be able to go through the edge F, only when a signal on the *sync_channel* channel is received, the guard condition of edge F must be true.

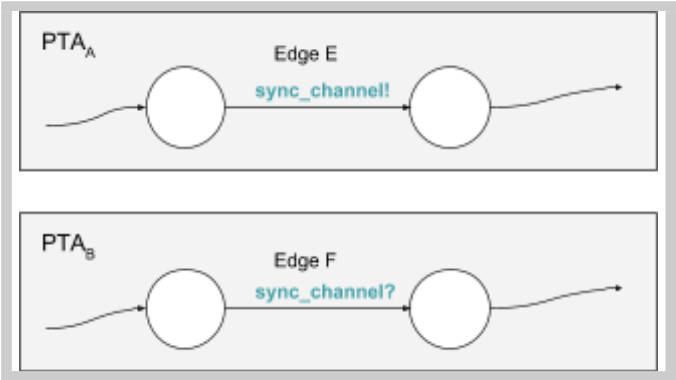


FIGURE 6.1

The syntax of a declaration of a channel with the name "*sync_channel*":
chan *sync_channel*;
PTA_A is the origin of the signal, it will use the *synchronization* attribute of the edge E, with the name of the channel and "!": *sync_channel!*.
PTA_B is the receiver of the signal, it will use the *synchronization* attribute of the edge F, with the name of the channel and "?": *sync_channel?*.
The passage of signal does not delay signal originator PTA. If nobody is waiting for a signal, it will be sent anyways without delays, and lost. Also, nobody will be able to receive it later.
There are two types of channels: 1. Regular channels in UPPAAL can connect one PTA that sends a signal to one PTA that receives it. 2. Broadcast channel can connect one PTA that sends a signal to many PTAs that receive the signal on the same channel.

6.1.4 *Clocks*

Clocks are variables that have a special property of growing at the same rate with time. They can be used by a node's *invariant* or an edge's *guard* (*enabling conditions*). *Clocks* can be compared one to another or compared to *integers* values. *Clocks* can be reset to an *integer* value. *Clock* value cannot be read into *integer* variables.
Definition:

```
clock  clock_1;
```

Condition:

```
clock_1 >= 5
```

Reset:

```
clock_1 = 0
```

6.1.5 Functions

Functions in UPPAAL are similar to functions used in any programming language; they can receive arguments, make calculations with variables, use loop structures, and conditional branching, also functions can return values. Functions are used by edges for update purposes, basically,

they are extensions of resetting abilities of discrete transitions of PTA.
Example:

```
int func( int a ){
        return 2 * a;
}
```

6.1.6 Branch point

Branch points represent probabilistic transitions. Each probabilistic edge has an *integer* representing its weight, and the probability of edge is a ratio between its weight and sum of all weight outgoing from the same branching point.
Example:
Probability 0.3 on edge going to u1, and 0.7 on edge going to d1.

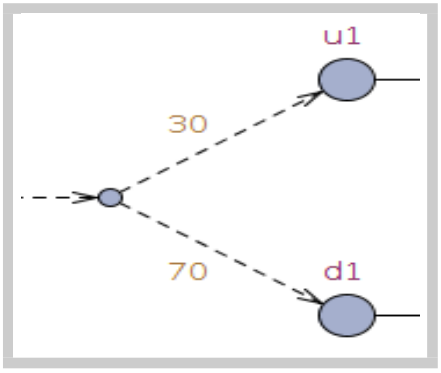


FIGURE 6.2

6.1.7 PTA templates

PTA templates are a well-defined Probabilistic Timed Automata with arguments. When arguments values are set, a template becomes regular PTA, which can be run by the system.
Each template has the properties: 1. Name; 2. List of variables that can be defined by value (copy of value) or by reference (influence on the actual object passed as an argument); 3. Local declarations of functions and variables. 4. Definition of the PTA as a graph.

6.1.8 Declarations

Declarations are a global code section shared by all PTA, which include common structure, variables (booleans, *integers*, *clocks*, channels) and functions.

6.1.9 System

The system is the code section that includes a definition of processes representing instances of templates (with arguments set to certain values), and initialization of all the processes that would

be running in the system.

6.1.10 Run time distribution

UPPAAL cannot natively represent any arbitrary run time distributions. Therefore we have to use an approximation to support all the run time distributions that are supported by PLP but not natively supported by UPPAAL. First, we will see all the run time distribution supported by UPPAAL and then introduce our approximation technique to implement any run time distribution.

Natively UPPAAL support an exponential distribution with an exponential_rate parameter. We can also specify the maximum time for transition, in case we know the maximum possible time for transition. Both rate of exponential and invariant specified in a location and shared between all outgoing transitions.

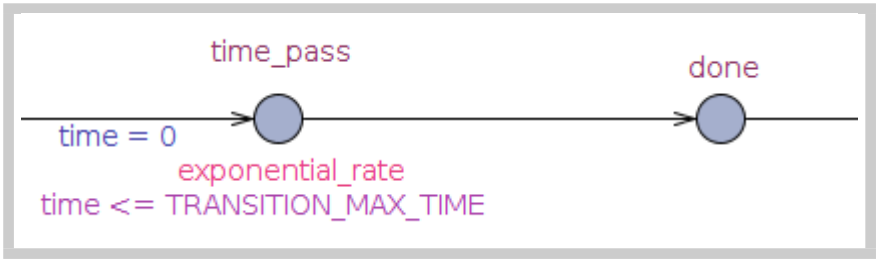


FIGURE 6.3

UPPAAL support a uniform distribution. Here we define a transition with uniform distribution in an interval [MIN_TIME, MAX_TIME]:

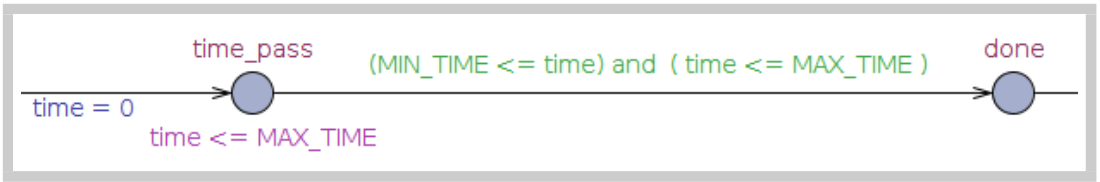


FIGURE 6.4

We define guard condition on a transition between *time_pass* and *done* states: “(MIN_TIME <= time) and (time <= MAX_TIME)”, this must happen between MIN_TIME and MAX_TIME time units.

Unfortunately, PTAs do not provide an easy way to represent more complex time distributions. To overcome this limitation, we will use an approximation to any possible run time distribution, by using a discretization of distribution. We will take the continuous distribution function and slice its range into smaller ranges; each small range will be represented by a uniform distribution of the average run time. For example normal distribution with mean 5 and standard deviation 1:

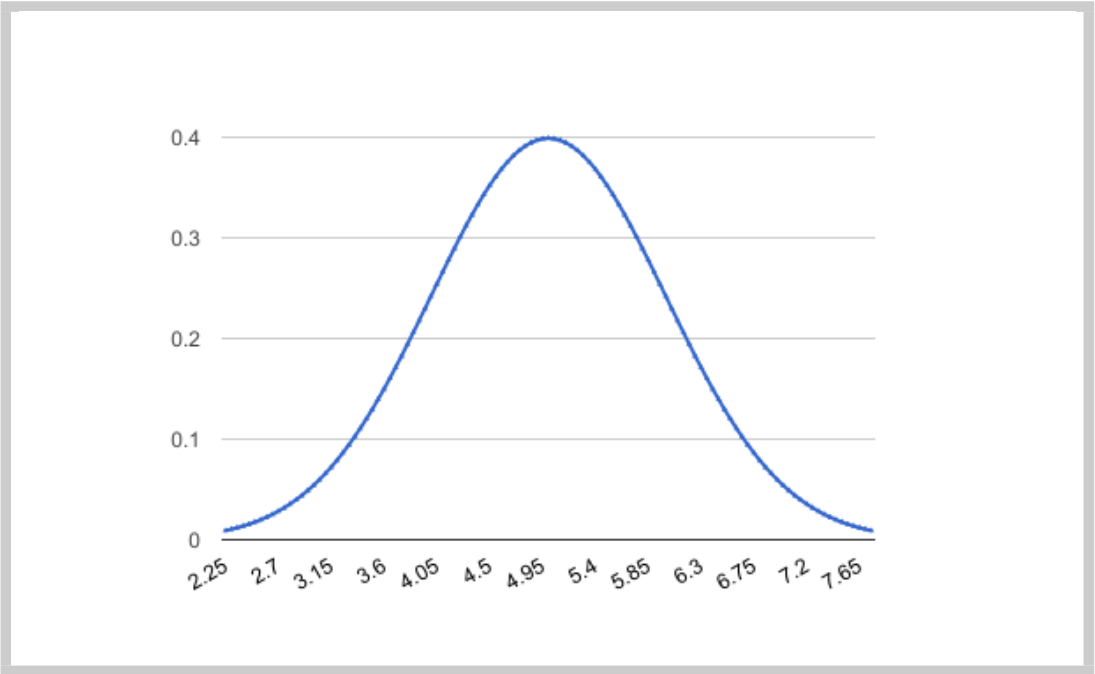


FIGURE 6.5

We will slice the range around the mean value into equal ranges. For each range, we will calculate the average value of the ends of the distribution function, and it will represent the discretized function value of this range. The discretized function is an approximation that can be improved with more slices of range. Here we can see the discretized distribution function on top of the original distribution:

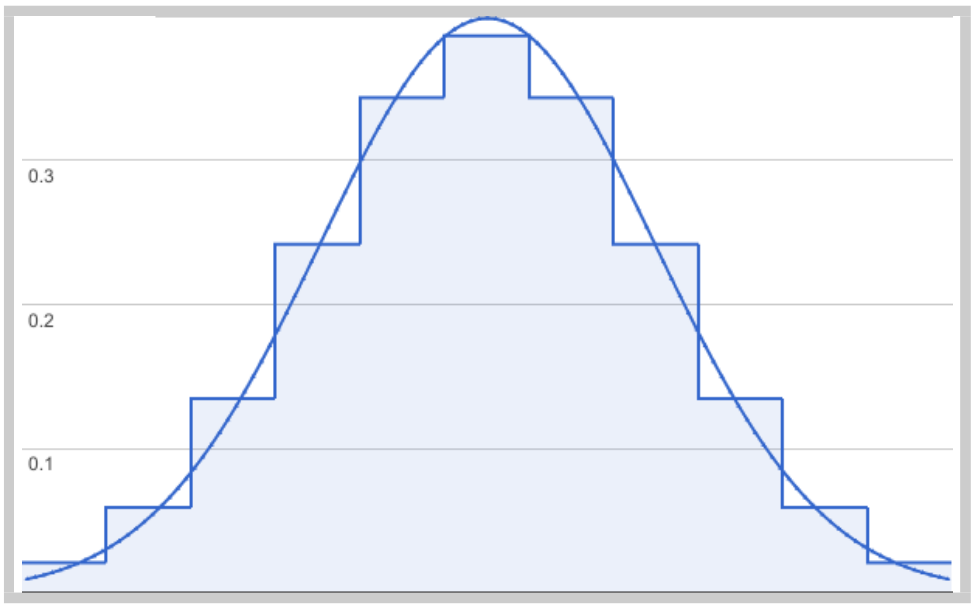


FIGURE 6.6

Each constant segment of the range is a uniform distribution that can be represented with PTA as we have seen. Now we will combine all segments into single PTA.

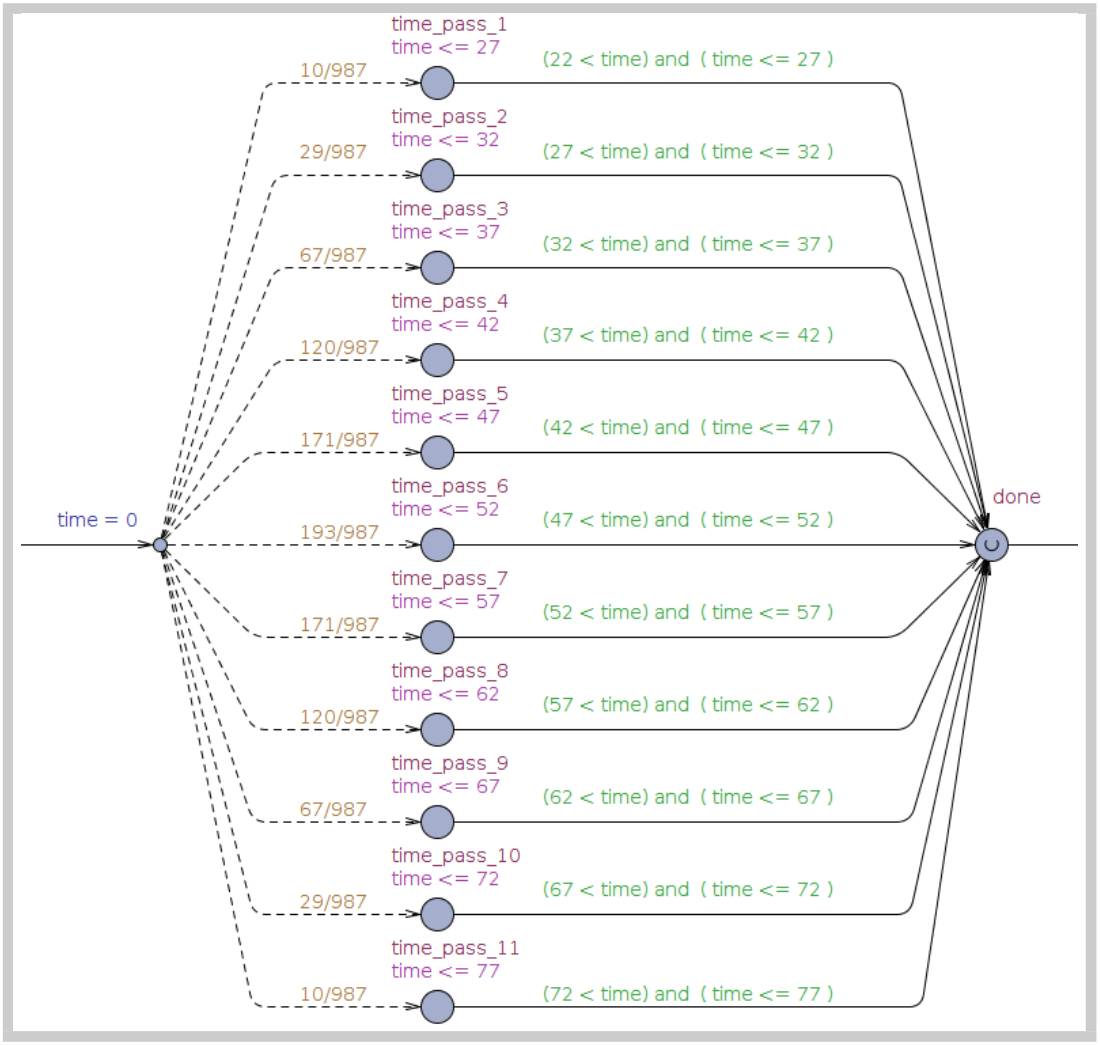


FIGURE 6.7

The area between the X-axis and the function is close to 1, similarly to the original normal distribution. We calculated the area of each segment, which is equal to the probability to choose this segment, then multiplied it by a constant factor (1000 in the example) and truncated the fractions; these are the probabilities (10/987, 29/987, 67/987, 120/987, ...) corresponding to probability to choose each segment. Each segment representing a certain run time, that we multiplied by 10, to increase the time granularity. The run time of each segment distributed uniformly over its range. If we execute this PTA many times (99,030 in the following example) we will see that we indeed produced a run time distribution very similar to the expected discretized distribution function:

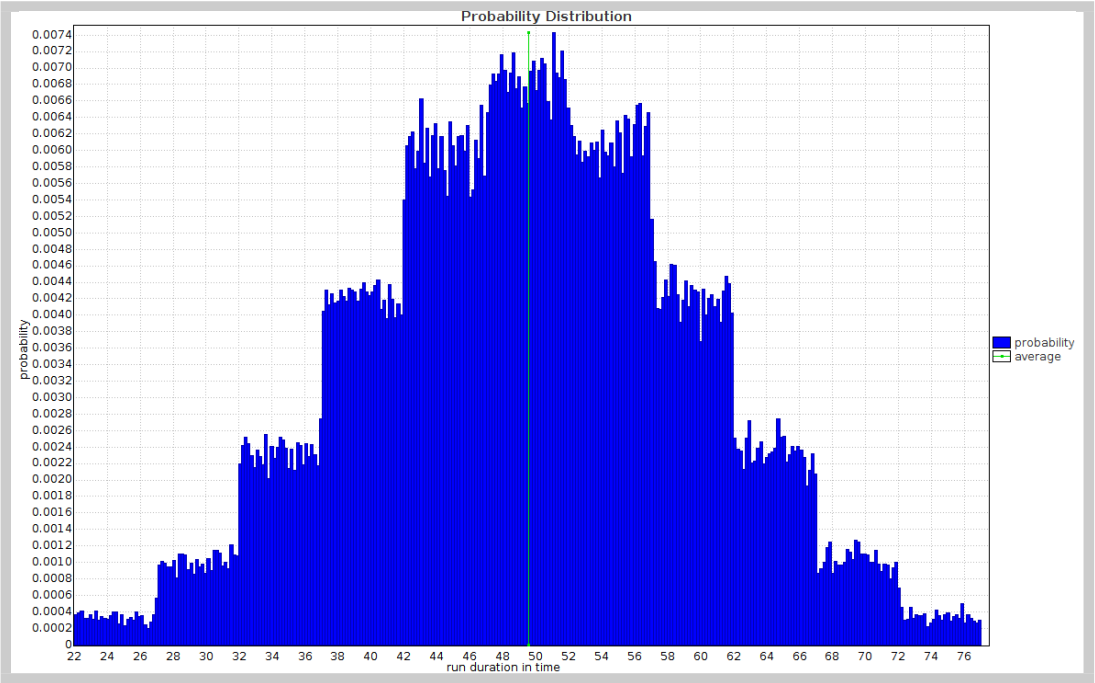


FIGURE 6.8

Examination of the area of each time segment corresponding closely to the probability of choosing this time segment in PTA, as we expected.
We will apply this technique of discretization of the distribution to other complex distribution used by PLPs.

6.1.11 Queries [9]

The queries are temporal propositions regarding system state, that can be true or false. UPPAAL can verify whether they are always true or false, and approximate the chance that they are true if only occasionally true.
Queries combine single temporal quantifier as specified below, with conditions on variables, *clocks* and reached states.

Name	Property	Explanation	Equivalent to
Possibly	*E<> p*	If exist a path that *p* is eventually true.	
Invariantly	*A[] p*	If for all paths *p* is true all the time.	*not E<> not p*
Potentially always	*E[] p*	If exist a path that *p* is true all the time.	
Eventually	*A<> p*	If for all the paths, *p* is eventually true.	*not E[] not p*
Leads to	*p - -> q*	For all the paths, if *p* is true, *q* will be eventually true.	*A[] (p imply A<> q)*
Probability eventually	*Pr[] (<>q)*	Calculate the probability of *q* eventually true.	
Probability always	*Pr[] ([]q)*	Calculate the probability of *q* always true.	

Example:
Below we can see a query that is true if eventually: variable a equals 2, *clock* time_x is not greater than 5, and the process process_x is reached its state *done*.

E<> ((a == 2) and (time_x <= 5) and (process_x.done))

6.2 Implementation of Formal Semantics in UPPAAL

6.2.1 Variables

There are 3 types of variables supported by PLP: 1. *Boolean* - *true* or *false*. 2. *Reals*. 3. *Integers*. The UPPAAL PTA model supports only discrete variables so booleans and *integers* can be easily represented. But we cannot use double precision floating point numbers in a straightforward way. To overcome this problem, we convert *reals* to *integers* with truncation. To minimize the loss of precision, first we will multiply the value of a *real* number by a power of 10 (i.e., precision factor), and then truncate the fraction part. For example, if we had variable with value 1.52376, and the precision factor is 10^2 , we will calculate 1.52376×10^2 , and after truncation, we will store the number 152 .

The change of values of *real* variables in PTA, to increase the precision as we just described, presents a problem with consistency of *real* variables values with *integer* variables. *Real* variables

may be compared or be assigned values according to the values of some *integer* variables. Therefore we must adjust the values of *integers* by the same factor as *reals* throughout the whole system to prevent the inconsistency.

Additionally, the adjustment of values of *reals* and *integers* produces an inconsistency with run time and values of constants, that may be used with variables that were adjusted. Therefore we have to update run times and constants accordingly to the same precision factor that we use for *reals* and *integers*.

Generally, variables in PLP can be defined to belong to a union of disjoint continuous ranges. Due to possible over complication of generated PTA, we support only a single range. For some variable X ∈ [MIN, MAX], we make sure with the *concurrent_write* function that for any write attempt with value V to X: MIN ≤ V ≤ MAX.

6.2.2 Conditions

The term *condition* is used in two different meanings in PLP: 1. As a typical condition, for example, a PLP precondition: *"a = b"*, which is true when the value of *a* equals the value of *b* then *"a = b"* holds. 2. As a requirement that needs to be made true by an appropriate assignment. For example, the PLP *goal condition*: *"a = b"* that requires that the values of a and/or b be changed somehow so that the condition will become true. Thus, the first is a condition statement, and the second is similar to an assignment, although it does not specify directly how the variables should be assigned.
The *condition* can be represented as a context-free grammar:

CONDITION → PREDICATE | FORMULA | NOT | FOR_ALL | EXISTS | AND | OR
VAR → [a-zA-Z]VAR_REST
VAR_REST → [0-9a-zA-Z_]VAR_REST | ε
A, B → VAR | CONDITION
NUM → [1-9]NUM_REST
NUM_REST → [0-9]NUM_REST | ε
VARS → VAR | VAR, VAR
PREDICATE → predicate_VAR(VARS)
MIN_NUM, MAX_NUM → NUM
MIN_INCLUSIVE, MAX_INCLUSIVE → true | false
OPERATOR → = | != | < | <= | > | >=
FORMULA → formula(A,NUM,OPERATOR) | formula(A, MIN_NUM,MIN_INCLUSIVE,MAX_NUM,MAX_INCLUSIVE)
NOT → not(A)
FOR_ALL → (∀,VARS,CONDITION)
EXISTS → (∃,VARS,CONDITION)
OR → ((A) or (B))
AND → ((A) and (B))

- Predicate - We can think about predicate and its fields as a boolean variable that can be true

or false at any given time. So we will define such boolean variable for each unique set of predicate and fields appearing in PLP. We can use it in a condition or assign a value to it.

- Formula - There are two kinds of formulas, the first is: *"Variable operation Value"*, for example: *"x <= 5"*; the second form is: *"$Value_1$ <= Variable <= $Value_2$"*, for example: *"2 <= y <= 4"*. Both of the forms can easily express a condition, but the assignment is more problematic. There are many possible assignments that make the condition true, and we cannot assign them all without making PTA impractical. However, it is possible to use nondeterministic transitions if the range of the variable is small, for example: *"0 <= x <= 2"*, so update attribute for edges would look like this:

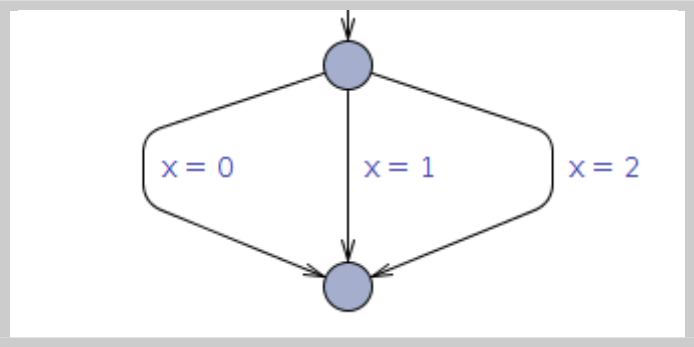


FIGURE 6.9

Generally, in such ambiguous cases, we will present a warning message regarding this ambiguity, but we will also generate a single value assignment. If it is a range, we will assign the value of the middle number, and if it is a comparison to number, we will use the closest possible number. For example: *"0 <= x <= 2"* will be converted to *"x = 1"*, and *"3 < x"* will be converted to *"x = 4"*.

- AND - As a condition, the statement will remain the same. As an assignment, all parts will be assigned with true values. For example: *"a and (4 < b)"* will become *"a = true, b = 5"*.
- OR - As a condition, the statement will remain the same. As an assignment, we will create an edge with a single assignment for each of the operands of NOT to be true. For example: *"a or (4 < b)"* will become *"Edge #1: a = true, Edge #2: b = 5"*.
- NOT - In comparison case, we will use logical negation. In a case of an assignment, we will need to assign everything inside with false value. For example: *"not(b)"* will be converted to *"b = false"*. If statement inside the Not is more complex than just a variable, we will assign the false recursively. For example: *"not(a or (4 < b))"* will become *"a = false, b = 4"*. If the application of Not creates an ambiguity, we will warn the user and assign the values by our assignment policy.
- FOR ALL - FOR ALL is a challenging quantifier to be represented by PTA. In a case of assignment, we would need to assign all the possible values to a variable. The only way to make sure all values are assigned with a certain edge, is to duplicate the whole network of PTAs for each assignment, which is not practical. Because of this problem, our system does not support the FOR ALL quantifier. If For All is found, we will warn the user and abort the compilation.
- EXISTS - We ignore the EXISTS quantifier both in comparison and assignment cases.

6.2.3 Success and failure probabilities

In UPPAAL instead of writing probabilities as fractions, we have to use *integer* weights. We convert the fractions to have a common denominator, and then we write the numerator of the fraction on top of an edge, as its weight, the denominator is present only implicitly as the sum of all outgoing weights.

6.2.4 Side effects

If the side effect is some formula, we have to know the values of all the variables, and if the values are unknown, we will produce an error message and abort the compilation.

6.3 *Control graph* compilation to network of PTAs

The *control graph* is based on four types of nodes: 1. Probabilistic. 2. Conditional. 3. Concurrent. 4. Sequential. We create a template PTA for each node type that represents its behavior. The *control graph* is a network of PTA_{Node} representing nodes that communicate with each other, schedule execution of each other and execution of available PTAs_{PLP}. Each node knows its direct predecessors *(1, … , j)*, which are using agreed variables to communicate with PTA_{Node} and signal when it can run. The PTA_{Node} is waiting in the *init_node* state for its direct predecessors to finish, and then signal on the *try_to_run* channel, then the PTA_{Node} checks whether running condition is fulfilled, one of two options: 1. All direct predecessors concluded, logical AND (&&) on all direct predecessors completion variables. 2. Any direct predecessors concluded, logical OR (||) on all direct predecessors completion variables.

Nodes in the *control graph* can update values of variables; variables that are used both by PLPs and other nodes in a *control graph*. Due to our obligation to respond without delays to *termination conditions* and continuously watch changes of variables for *concurrent conditions*, we must use the *concurrent module* for an update of every variable. In the following implementations, we reserve the states: *update, update_1, …, update_k* to constitute placeholders for access to the *concurrent module*; that would write to variables with the *concurrent_write* function.

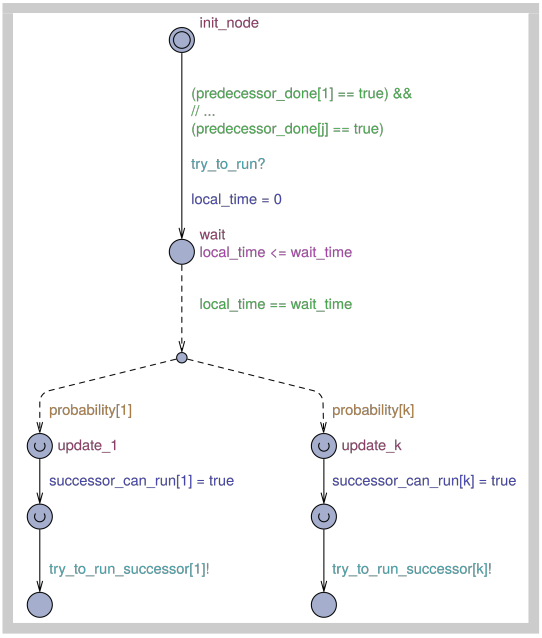
Now we will show the implementation of each type of PTA_{Node}.

6.3.1 *Probabilistic nodes*

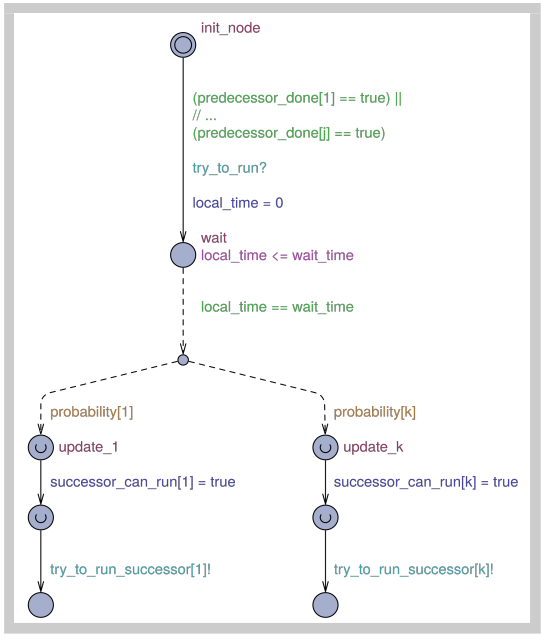
When the condition to run is satisfied, PTA_{Node} waits in a *wait* state, for a time interval determined by *control graph*. Then the PTA_{Node} chooses a single path to follow, out of the possible k paths, according to the specified probability of each path. Then the PTA_{Node} updates *successor_can_run[i]* (i∈[1..k]) for successor node to run. Eventually, signals to the successor on *try_to_run_successor[i]* channel to check if it can run.

PTA_{Node} template:

Wait for All direct predecessors:



Wait for Any direct predecessors:

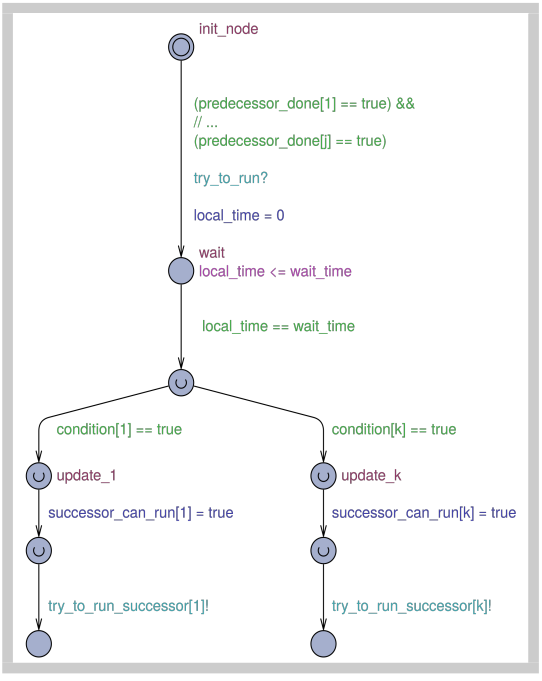


6.3.2 *Conditional nodes*

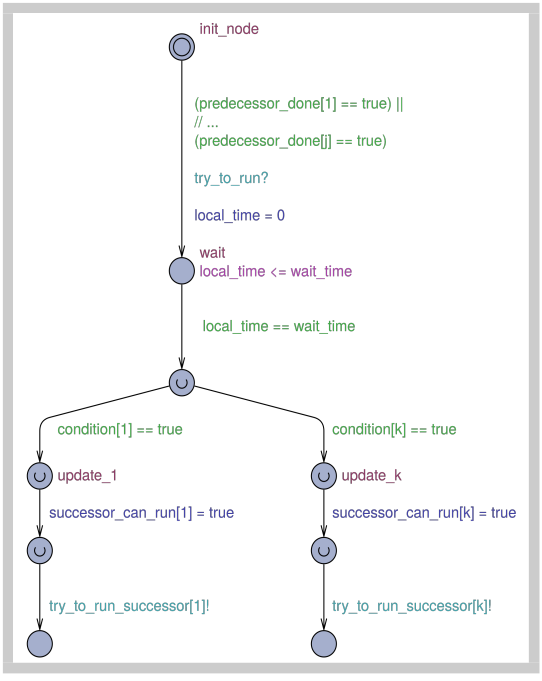
When the condition to run is satisfied, PTA_{Node} waits in a *wait* state, for a time interval determined by *control graph*. Then the PTA chooses a single path to follow, out of the possible k paths, by conditions on edges. In the PTA_{Node} template below the conditions are: *condition[1] == true*, …, *condition[k] == true*. Then the PTA_{Node} updates *successor_can_run[i]* (i∈[1..k]) for chosen successor node. Eventually, signal to successor on *try_to_run_successor[i]* channel to check if it can run.

PTA_{Node} template:

Wait for All direct predecessors:



Wait for Any direct predecessors:

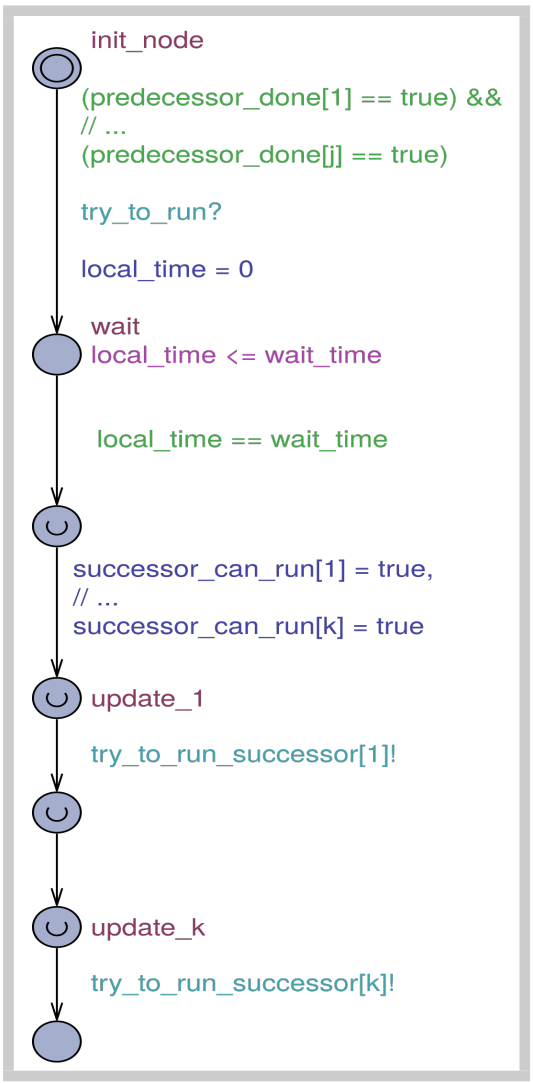


6.3.3 *Concurrent node*

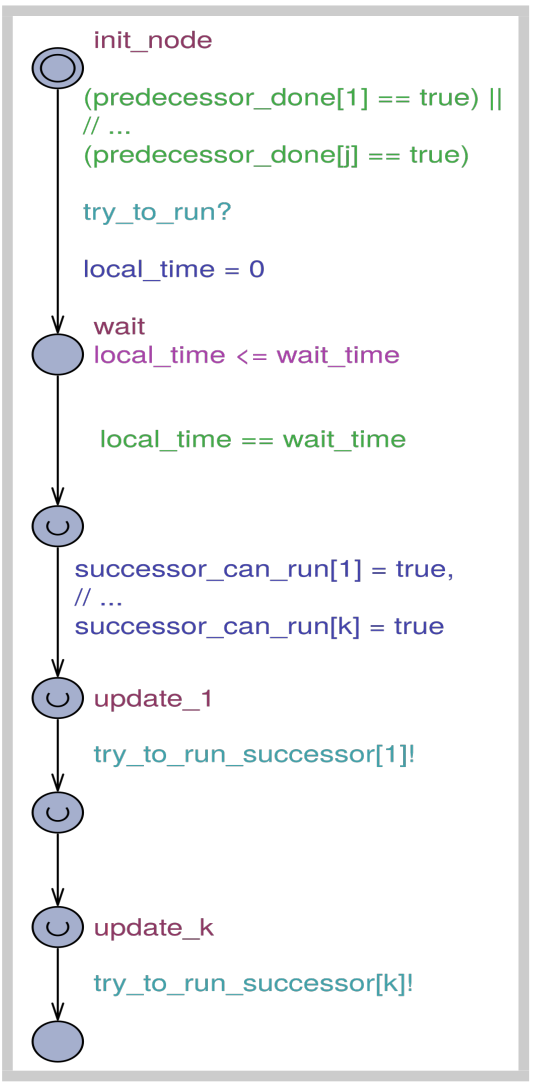
When the condition to run is satisfied, PTA_{Node} waits in a *wait* state, for a time interval determined by *control graph*. Then the PTA_{Node} updates *successor_can_run[i]* ($\forall i \in [1..k]$) for all successors nodes. Then the PTA_{Node} signals to each successor on *try_to_run_successor[i]* channel to check if it can run. We should notice that other nodes receive signal on *try_to_run_successor[i]* channel in a sequential manner. If their run condition fulfilled and they can run, they will only pass the synchronization edge, rest of their executions can be scheduled later, regardless of the synchronization edge. For example, if node id_1 passed synchronization first, and then node id_2, the next transition can be scheduled to be by id_2 or id_1, and each execution can produce different scheduling combinations.

PTA_{Node} template:

Wait for All direct predecessors:



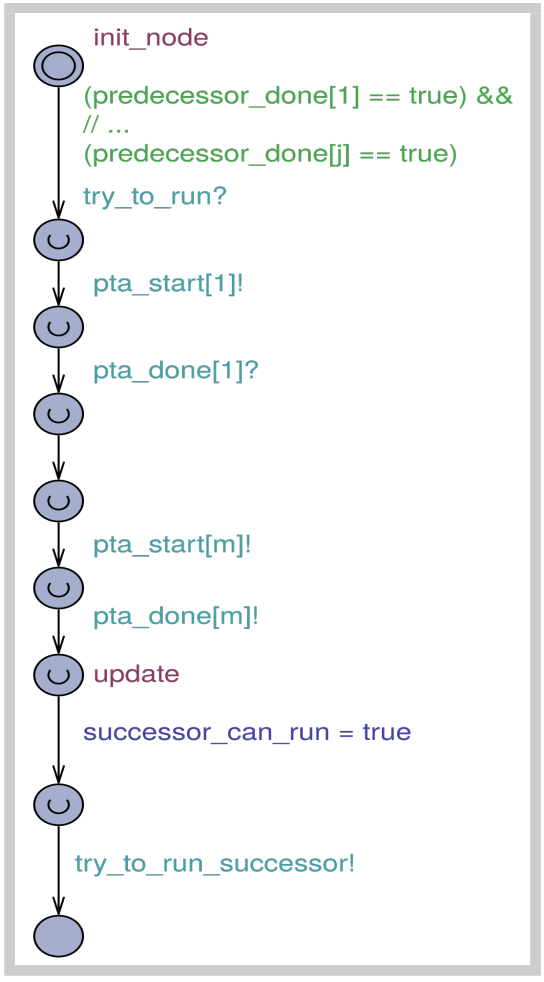
Wait for Any direct predecessors:



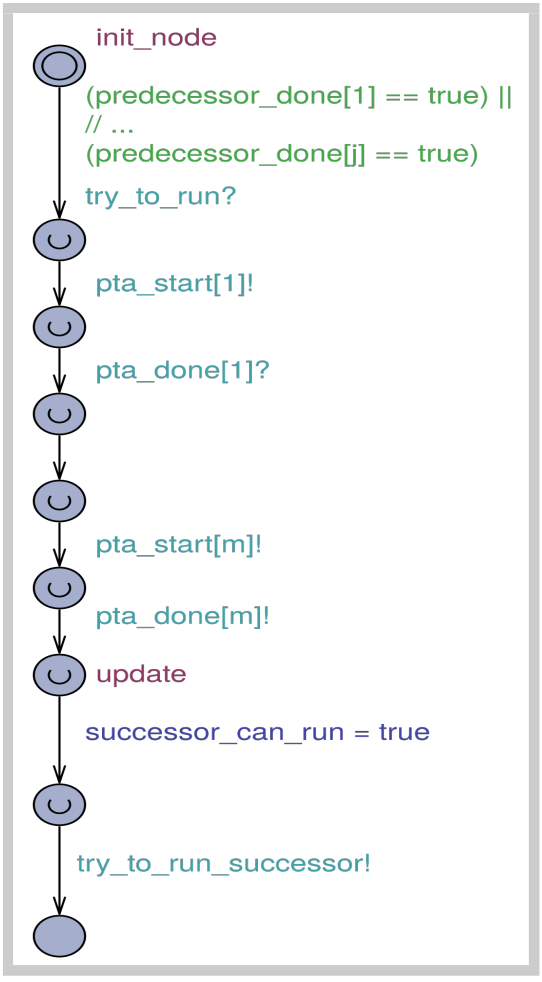
6.3.4 Sequence of $PTAs_{PLP}$

When the condition to run is satisfied, PTA_{Node} goes over the sequence of $PTAs_{PLP}$ it should execute, and signals each PTA_{PLP} in a time to run with *pta_start[i]* ($\forall i \in [1..m]$) channel, then waits for PTA_{PLP} to complete running on *pta_done[i]* channel. When all $PTAs_{PLP}$ finished running, PTA_{Node} updates *successor_can_run* for successor PTA_{Node} to run. Eventually, signals to successor PTA_{Node} on *try_to_run_successor* channel to start running if possible.
PTA_{Node} template:

Wait for All direct predecessors:



Wait for Any direct predecessors:



CHAPTER 7

Control graph examples

We will look at three examples of a *control graph* involving an autonomous robot with an arm.

7.1 PTAs$_{PLP}$ executions and conditions

We need to move the robot from room A to room B, through doorway 1, the door can be either open or close, but it is definitely unlocked.

7.1.1 Available PTAs$_{PLP}$ with self-explanatory names:

- move_to(doorway|target|through_doorway) - Achieve PTA$_{PLP}$.
- is_door_open - Observe PTA$_{PLP}$.
- open_door - Achieve PTA$_{PLP}$.

7.1.2 Initial state:

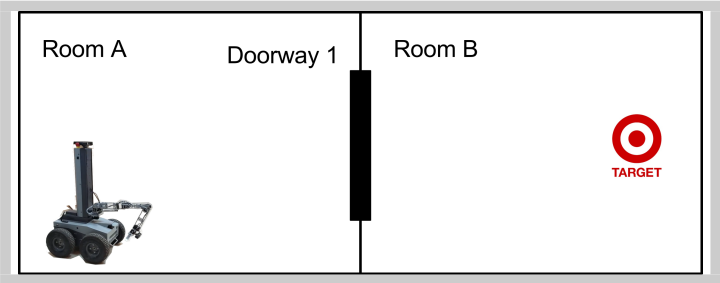


FIGURE 7.1

7.1.3 *Control graph*:

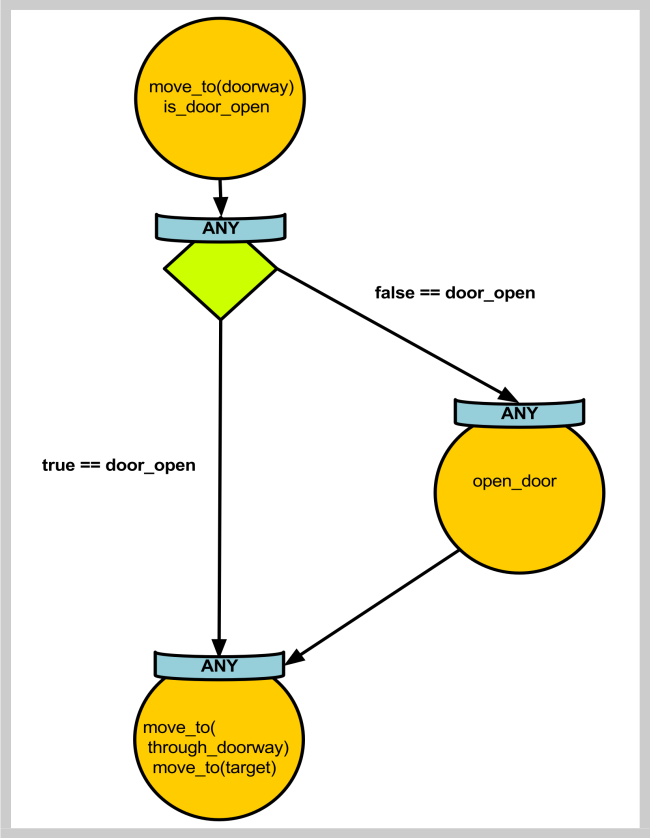


FIGURE 7.2

7.1.4 Goal state:

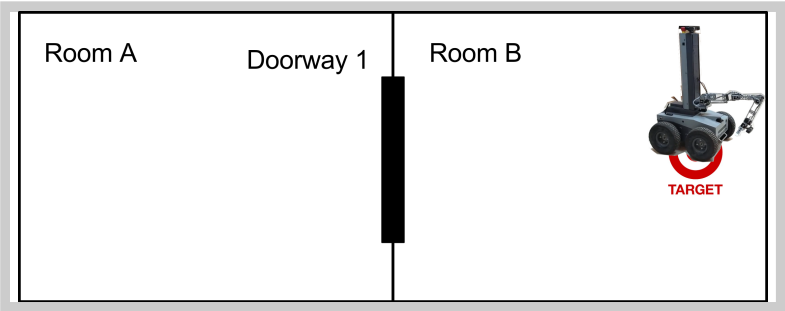


FIGURE 7.3

7.2 Concurrent execution

We need to move the robot from room A to room B, through doorway 1, the door can be either open, closed or locked.

7.2.1 Available $PTAs_{PLP}$ with self-explanatory names:

- move_to(doorway|target|through_doorway) - Achieve PTA_{PLP}.
- is_door_open - Observe PTA_{PLP}.
- is_door_locked - Observe PTA_{PLP}.
- key_take - Achieve PTA_{PLP}.
- key_hold - Maintain PTA_{PLP}.
- door_open - Achieve PTA_{PLP}.
- door_unlock - Achieve PTA_{PLP}.

7.2.2 Initial state:

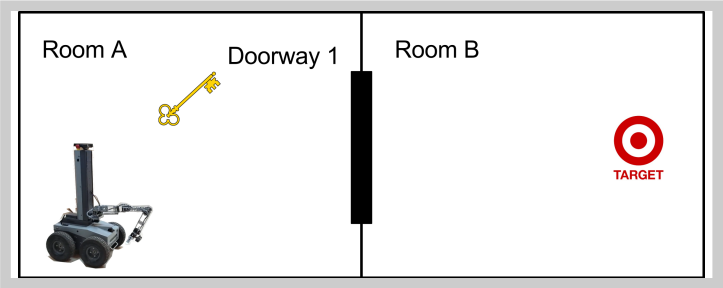


FIGURE 7.4

7.2.3 *Control graph*:

The robot will move to a key and pick it up with its arm; then it will move to the doorway. It will check if the door is open, if the door is open robot can move through the doorway, and to the target. If the door is closed, the robot will check whether it is locked, if unlocked, it will open the door and continue. If the door is locked, the robot should unlock it; it is possible only if the key_hold PTA_{PLP} is running and has not failed, then the door can be unlocked and opened as before.

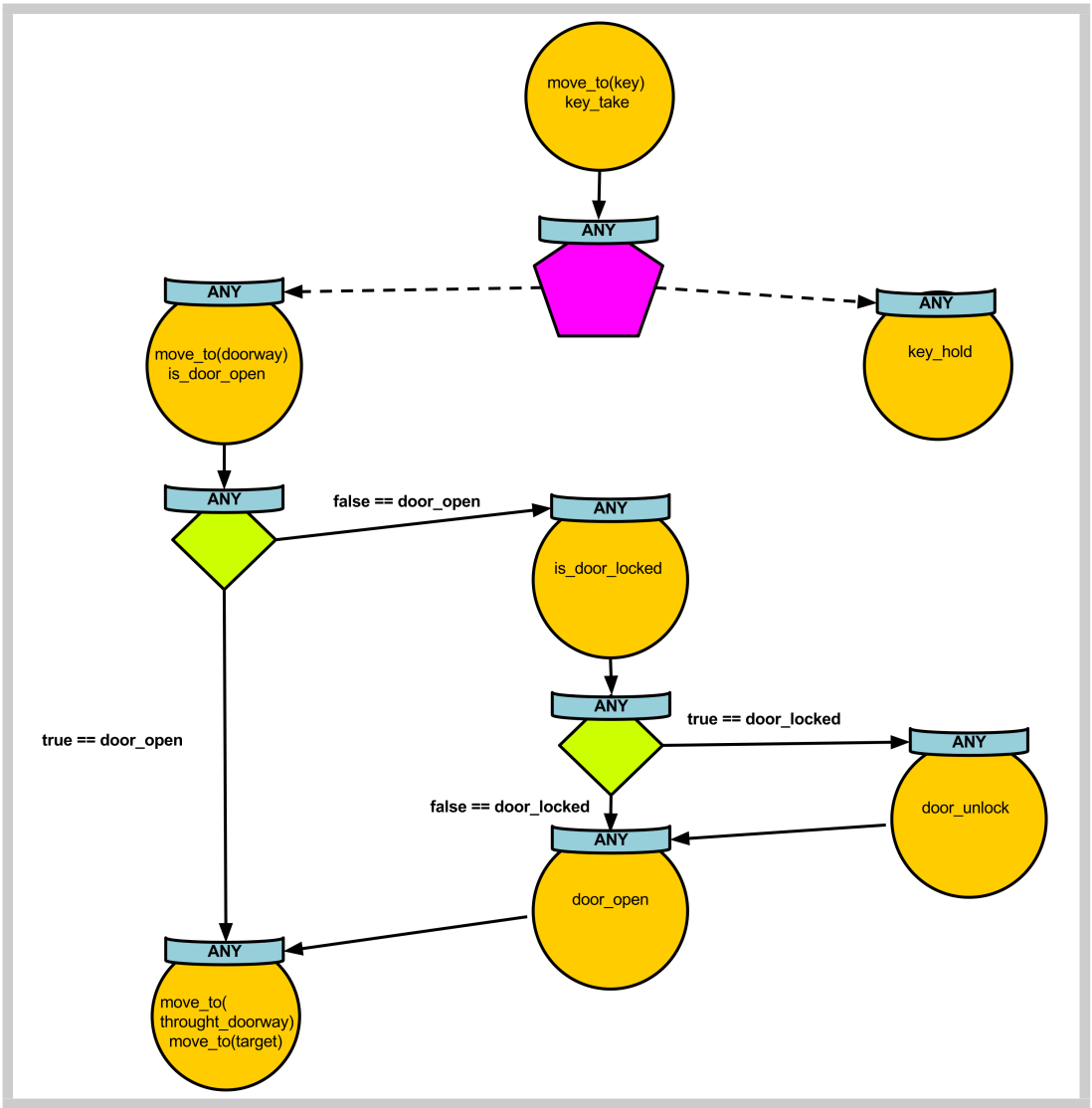


FIGURE 7.5

7.2.4 Goal state:

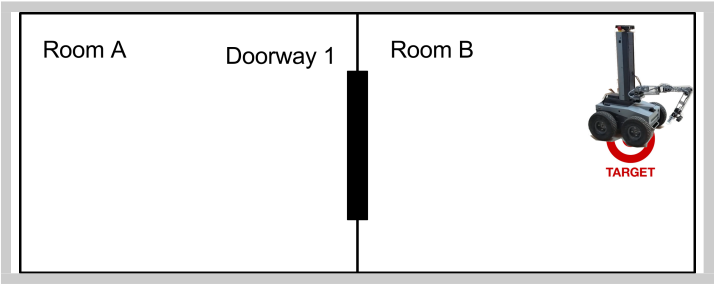


FIGURE 7.6

7.3 Probabilities

We need to move the robot from room A to room B or C, through doorway 1 or 2, the door can be either open or closed, but it is definitely unlocked.

7.3.1 Available $PTAs_{PLP}$ with self-explanatory names:

- move_to(doorway_1 | doorway_2 | target_1 | target_2 | through_doorway_1 | through_-doorway_2) - Achieve PTA_{PLP}.
- is_door_open(door_1 | door_2) - Observe PTA_{PLP}.
- door_open - Achieve PTA_{PLP}.

7.3.2 Initial state:

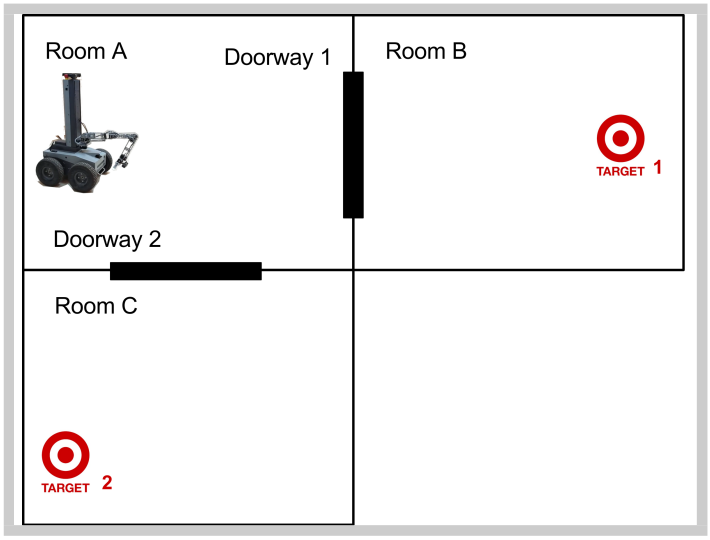


FIGURE 7.7

7.3.3 *Control graph*:

The robot will choose one of the rooms, room B with probability 75% and room C with probability 25%. The robot will move to the doorway, will open the door if it is closed, move through the doorway to the target.

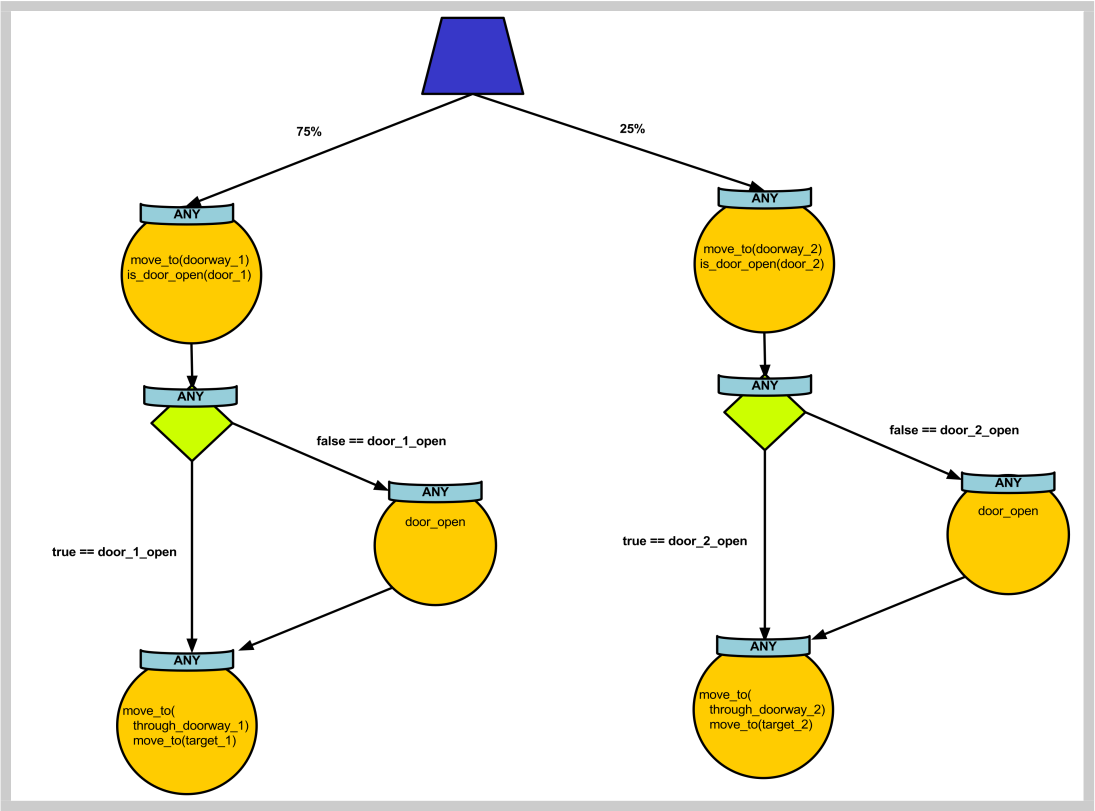


FIGURE 7.8

7.3.4 Goal states:

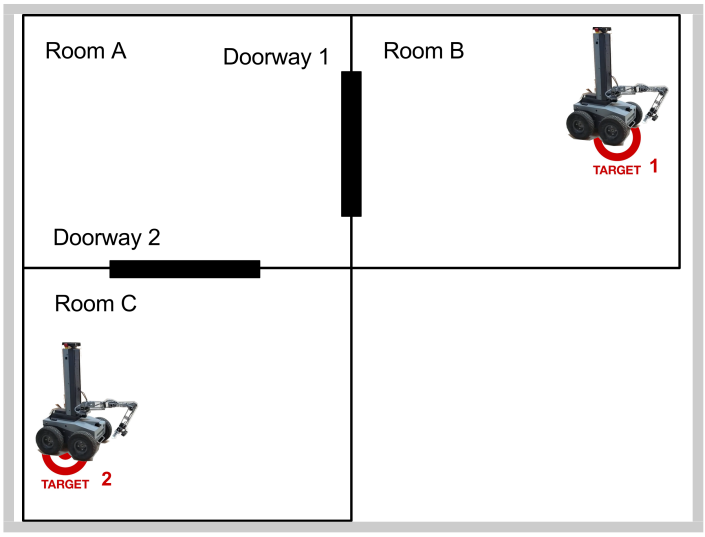


FIGURE 7.9

CHAPTER 8

Software

Our software is based on the software developed by Brafman, Ronen I., Michael Bar-Sinai, and Maor Ashkenazi [5]. The software receives the arguments: definitions of PLPs, *control graph*, and configuration file, in XML format. The software uses implementation of the methods presented in this thesis to produce a system of PTAs in the format used by UPPAAL.

8.1 Software setup

The software available from github: https://github.com/a-l-e-x-d-s-9/plps_verification.git
The code can be edited with any Java editor, we have used IntelliJ IDEA editor. Project files included in the git repository, project: CodeGenerator, should be compiled as an application with main class: *codegen.common.CodeGenerator*. The project has to be linked with Apache Commons Math 3.6.1 that is also included in the git repository.
Execution:
java -jar CodeGenerator.jar -verify [1:plp-folder-path] [2:control-graph-file] [3:generate-uppaal-system-file] [4:configuration-file]
Arguments:
[1:plp-folder-path] - folder with all the XML files corresponding to PLP specifications.
[2:control-graph-file] - XML file defining *control graph*.
[3:generate-uppaal-system-file] - the UPPAAL system file that would be generated by the software.
[4:configuration-file] - XML file containing all the configurations.
Generated UPPAAL system tested with UPPAAL 4.1.20-stratego-4 (rev. 7), November 2016 32bit version on Linux: Ubuntu 17.04.

8.2 Brief introduction to the software

8.2.1 Configuration file

Configuration file defined by XSD file in Appendix A2.1. The file containing: 1. Set of settings defining a few generation parameters. 2. Initial value and range of variables, parameters,

constants, and resources.
The available settings:

- “precision_multiplier_for_numbers_and_time” - the precision factor that we use to convert all values provided by PLP into values used by the UPPAAL system. Any real or *integer* variable, resource, constant or parameter in PLP, would be multiplied by this number and the fraction would be ignored.
- “observe_variable_samples” - the maximum amount of samples that any *Observe PLP* will produce to simulate observation of some variables. Samples will be uniformly taken from the range in which *observed variable* is defined in.
- “run_time_amount_of_intervals_for_discretization” - this setting defining that amount of intervals used to approximate normal distribution and gamma distribution.

To initialize value or/and range inside which variables, parameters, constants, and resources are defined, we can use their name from PLP. Parameters must also have the PLP name they belong to. Generally, for any variable, we can define an initial value, minimum and maximum boundaries of the range, and whether the variable should be used exclusively by *Maintain PLP* so it can not be written by two different *Maintain PLP*s at the same time or not.

8.2.2 *Control graph* file

The *control graph* file is defined by the XSD file in Appendix A2.2. It defines the root node of the *control graph* and all the available control nodes as we discussed.

8.2.3 UPPAAL system generation

The generation of an UPPAAL system is done in a few steps. First of all, the software goes over all the provided PLPs, extracting all the variables available. Then it extracts the settings, values and ranges for variables, and parameters. Then it produces UPPAAL template for each PLP. The *control graph* is generated with references to PLPs. If there is any problem with provided files, the software will provide information about the problem. The generated UPPAAL system file can be imported into UPPAAL and used to verify the system. The software is designed to be friendly to a person who wants to verify PLPs and *control graph*. Therefore in the generated system, we use PLPs’ names as PTAs names. For every UPPAAL variable the original name from XML appears in a comment in the initialization function: *initialize_variables()*, inside PTA_{PLP} there are comments pointing to the origin of an action from PLP. The system makes sure that any variable that is being read with *concurrent_read* function has been written beforehand by *concurrent_write* function. Also, the system makes sure that for every write into a variable, if the variable defined inside a range, any value outside the range would not be allowed.

8.3 Limitations

The *integer* in UPPAAL defined on the range $[-32768, 32767]$. Due to our reliance on *integers* for *clocks* and variables, we are limited to values in acceptable *integer* range. The precision factor that is larger than 1, reduces the time intervals and variables we can simulate and verify even further. Therefore it is important to notice that with a precision factor of X, we are limited to up to $\frac{32767}{X}$ time units in the real world, and variables in the range $\left[0, \frac{32767}{X}\right]$ in the real world.

8.4 System example

Let us look at example 7.2. We implemented a simple representation for every PLP (Appendix A3.3), and defined a *control graph* (Appendix A3.1) and configuration file (Appendix A3.2). In this example, a robot needs to move from one room to another room through a door that may be open, closed, or locked. The robot can check if the door is locked and unlock it, and open a closed door.

We defined all the PLPs to use uniform distributions, with the following boundaries(in UPPAAL time units):

PLP	Lower boundary	Upper boundary
achieve_door_open	100	200
achieve_door_unlock	100	200
achieve_key_take	100	200
achieve_move_to	400	600
maintain_key_hold	?	?
observe_is_door_locked	300	700
observe_is_door_open	300	1000

Control nodes defined not to have any delays, so the time can pass only inside PLPs.
There are four possible execution paths for this *control graph*, if we look only on executed PLPs (shared cells between different columns represent the same execution of PLP):

<table>
<tr><td>Path 1:
Parallel to other paths, makes sure that robot holding a key</td><td>Path 2:
In case the door is open</td><td>Path 3:
In case the door is closed but unlocked</td><td>Path 4:
In case the door is closed and locked</td></tr>
<tr><td colspan="4">move_to(key)</td></tr>
<tr><td colspan="4">key_take</td></tr>
<tr><td rowspan="6">key_hold</td><td colspan="3">move_to(doorway)</td></tr>
<tr><td colspan="3">is_door_open</td></tr>
<tr><td rowspan="4">move_to
(through_doorway)</td><td colspan="2">is_door_locked</td></tr>
<tr><td rowspan="2">door_open</td><td>door_unlock</td></tr>
<tr><td>door_open</td></tr>
<tr><td>move_to
(through_doorway)</td><td>move_to
(through_doorway)</td></tr>
<tr><td>move_to(target)</td><td>move_to(target)</td><td>move_to(target)</td></tr>
</table>

In this example, a possible use for the system we generated and UPPAAL capabilities, would be the problem of verifying minimum safe duration of time to hold the key. The key is only needed for a single execution path, in case the door is locked. But the robot takes the key before it checks whether the door is locked. The robot does not need the key in case the door is open or unlocked, so the key can just be dropped as soon as robot checks that the door is locked. But for the sake of the example, we can assume that robot drops the key only as a result of execution completion of *Maintain PLP* key_hold. So we want to find the minimum amount of time robot must hold the key, for it to be used by other PLP to unlock the door. We can calculate when door_unlock PLP might need the key:

PLP	Run time range
move_to(key)	[400,600]
key_take	[100,200]
move_to(doorway)	[400,600]
is_door_open	[300,1000]
is_door_locked	[300,700]
door_unlock	[100,200]

Thus door_unlock can start between 1500 and 3300 time units, and run for 100 and 200 more time units. Therefore the key might be used between 1500 and 3500 global time units.
Now if we consider when key_hold PLP might start:

PLP	Run time range
move_to(key)	[400,600]
key_take	[100,200]

It would start between 500 and 800 time units. We can argue that because door_unlock can run up to 3500 time units after start, and key_hold can start as soon as 500 time units after start, 3500 - 500 = 3000 time units run for key_hold would be a safe estimation. But we should remember that both of the first PLPs executed between the key holding path and unlocking path are shared executions. We can combine time execution of move_to(key) and key_take and write it as X∈[500,800]. Therefore door_unlock execution frame is: [1000+X,2500+X], and key_hold starts after X time units and must run up to 2500+X time units, we can conclude that key_hold must run for at least 2500 time units, we add one more time unit to run longer than door_unlock. To verify with UPPAAL the conclusion we have reached, we can define a run time boundaries for uniform distribution for key_hold, and test that whether the robot can always finish successfully. We use a query “For all paths eventually (*robot_location == at_b_target*)”:

A<> concurrent_info.concurrent_data[0].value == 500

- In the *initialize_variables()* function we can see that : *robot_location* received the variable ID 0, and the value of *at_b_target* converted to 500.

When key_hold run time set to [2500,2500], UPPAAL will answer that query does not satisfied, while for [2501,2501] the query is satisfied as we predicted.

CHAPTER 9

Related Work

One can argue that the foundation of the whole field of modern computer science relies on the formalism provided by Church–Turing thesis. The idea of providing a formal semantics for software development has been present and advocated for many decades [21]. There is a wide variety of formal verification tools and many possible ways to apply them throughout the whole cycle of hardware and software development. Here we will focus on a few similar attempts to our own, to describe robotic system specifications, formalize them and use for verification and validation.

Procedural Reasoning System (PRS) [13] is a high-level language for supervision and control systems for autonomous mobile robots. The key features of PRS: 1. Ability to construct and act on partial plans. 2. Ability to pursue goal-directed tasks while at the same time being responsive to changing patterns of events in bounded time. 3. Facilities for managing multiple tasks in real-time. 4. Mechanisms for handling stringent real-time demands of its environment. 5. Meta-level reasoning capabilities. A PRS kernel is composed of three main elements: 1. A database which contains facts representing the system view of the world and which is constantly and automatically updated as new events appear. 2. A library of plans (or procedures, or scripts), each describing a particular sequence of actions and tests that may be performed to achieve given goals or to react to certain situations. 3. A *tasks graph* which is a dynamic set of tasks currently executing. PRS interpreter follows a dynamic *task graph*, to achieve goals, by execution of procedures that may alter PRS belief database.

The goals in PRS are used as basic instructions of procedures; when combined with conditional instructions and loops they describe the inner algorithm of a procedure/module. Therefore goals in PRS are fundamentally different from PLPs; because each PLP describes the behavior of a single module as a whole and not its algorithm. Nevertheless, the types of goals supported by PRS are comparable to the existing PLPs types. PRS *achieve goal* - indicates that a certain statement is satisfied at some moment in time, analogous to a goal element of *PLP Achieve*. PRS *test goal* - implies that either the PRS belief database contains information regarding the statement or there is a procedure to find out what are the missing parts of the statement; *test goal* similar to the *observed variable* by *PLP Observe*. PRS *wait goal* - waits until a certain statement is true, is similar to *PLP Detect*. PRS *preserve goal* and *maintain goal* - passively and actively (respectively) make sure that some statement is true; they both akin to the maintained condition by *PLP Maintain*.

The *tasks graph* in PRS describes the hierarchy of procedure calls for currently executed procedures; it changes with time according to the execution flow. Therefore, *tasks graph* is principally different from our *control graph*, which describes a complete plan of all possible execution paths.

Languages based on the BDI (Belief, Desire, Intention) agent architecture as embodied in PRS and its variants define a whole class of PRS-like architecture [26] of agents. PRS-like agents can me mapped [27] to a *reachability graph* structure. A *reachability graph* is a set of asynchronous interleaving finite automatons; it essentially represents a simulation of multiple executions of a program, with nodes in the graph corresponding to execution states, and transitions in the graph corresponding to atomic action execution. A *reachability graph*s are used as the representation underlying the SPIN model checker [11].

Task Description Language (TDL) [23] for robot control was designed as an extension of C++, with task decomposition, synchronization, execution monitoring and exception handling. Task trees are the basic representation underlying TDL that encodes the hierarchical decomposition of tasks into subtasks, as well as synchronization constraints between tasks. Each task tree node has an action associated with it, which is essentially a parameterized piece of code. An action can perform computations, dynamically add child nodes to the task tree, or perform some physical action in the world. Also, actions can either succeed or fail. The node in a task tree can represent the goal that needed to be achieved, command that should be executed, monitor of state and exception handlers. Goal nodes can generate subtrees that need to be executed to achieve this particular goal; therefore task tree is highly dynamic structure. TDL is implemented using a compiler that transforms the task definitions into pure C++.

The nodes in the task tree resemble the nodes in the *control graph*, both support serial and parallel execution, and possess equivalent to PLP's repeat extension and a *termination condition*. The definite disadvantage of TDL due to inherent exclusive pairing with C++ code, constitutes very limiting approach in comparison to a universal acceptance of module implementations, as embodied by PLPs. One of the strong points of TDL and task trees is the handling of exceptions in a hierarchical structuring of exceptions; similarly to the state of art mechanisms of "catch and throw" used by many programming languages. In comparison, PLPs and *control graph* do not support the concept of exceptions at all.

Programs written in TDL can be automatically translated [24] into the SMV [7] model-checking language with which it is possible to perform a model checking using standard algorithms, and then translate counterexamples back into terms that are meaningful to the software developer. Symbolic Model Checking is a verification technology based on the exhaustive exploration of a system's achievable states. Given a model of a concurrent system, and the desired property of that system, a model checker will examine all possible executions of that system, including all possible interleavings of concurrent threads, and report any execution that leads to a violation of that property. The report is typically a counterexample to the property. This technique is making possible to verify important properties of TDL: check for deadlock, liveness, and absence of resource conflict.

Reactive Model-based Programming Language (RMPL) [12] is an object-oriented language Lisp-like programming language, that was developed for highly autonomous reactive systems, particularly in the aerospace domain. RMPL provides a framework for constraint-based modeling, as well as a suite of reactive programming constructs. RMPL based upon several basic constructs that allow parallel and sequential execution, conditions, loops, preemption of execution and probabilistic transitions. To support efficient execution or reasoning, RMPL code is compiled into hierarchical constraint automata (HCA).

There are many differences between RMPL and PLPs. RMPL is a programming language

the was designed with a strict model in mind; it is based on very simple constructs, that when combined can implement the functionality of a complex module. While PLP was designed to be a specification of the behavior of already implemented module; to describe its interfaces, influences on the world, probabilistic properties to succeed and run time limitations. Nonetheless, there are similarities between the constructs in RMPL and combination of PLPs and *control graph* features, that we will notice by going over the constructs. 1. PARALLEL - concurrent execution; an equivalent of the *Concurrent Node*. 2. SEQUENCE - sequential execution; an equivalent of the *Sequential Node*. 3. IF-THENNEXT-ELSENEXT, UNLESS-THENNEXT, and WHEN-DONEXT - conditional branching and temporal suspension; an equivalent of the *Conditional Node*. 4. WHENEVER-DONEXT - waits until a constraint is entailed, then execute expression, possibly repeatable; can be accomplished by interconnecting two *Conditional Nodes*. 5. DO-WATCHING - executes expression until completion or until a constraint is entailed; an equivalent of PLP's *termination condition*. 6. ALWAYS - iteration of expression; an equivalent of PLP's repeat extension.

RMPL constructs translated into hierarchical constraint automata (HCA) model and can be verified with *RMPLVerifier* [17]. *RMPLVerifier* uses greedy forward-directed search to find counterexamples to the program's goal specification the most likely executions that do not achieve the goal within a given time bound.

Behavior Interaction Priority (BIP) [2] is a framework for modeling heterogeneous real-time components. Every component has ports for communication with other components. Component created by a composition of three distinctive layers: 1. Behavior - set of transitions. 2. Interactions - between transitions of the behavior. 3. Priorities - used to choose among the possible interactions. The atomic components can be combined into composite components. Component construction allows preservation of properties of the underlying behavior. The characterization of components' transformations can provide correctness by construction and deadlock-freedom.

The Generator of Modules (G^{en}oM) [10] is a tool to design real-time software architectures [20] utilizing a formal description of a module. G^{en}oM separate a generic module into two levels: control level and functional level. Both of levels can be described by BIP [3]. Given module's formal description G^{en}oM can generate automatically code for a module and test program. Due to BIP integration, the model checker can be used to verify safety properties and deadlock-freedom of the generated module.

G^{en}oM offers a structured development context that allows the programmer to focus on its algorithms, without the need to care about the real-time process development and architecture design. The downside of G^{en}oM is the incompatibility with code and modules that were not designed specifically for it, contrary to PLPs' paradigm.

CHAPTER 10

Conclusions

We have introduced a formal semantics for PLPs with *Probabilistic Timed Automaton* templates representation for each PLP type. Also, we introduced and formalized robotic controller - *control graph*. Finally, we created a software that receives a set of PLPs and *control graph*, and generates the formalization we proposed here, as a concrete UPPAAL system that can be verified with a model checker UPPAAL.

Bibliography

[1] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[2] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 3–12. Ieee, 2006.

[3] Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Félix Ingrand, and Joseph Sifakis. Incremental component-based construction and verification of a robotic system. In *ECAI*, volume 178, pages 631–635, 2008.

[4] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126. IEEE, 2006.

[5] Ronen I Brafman, Michael Bar-Sinai, and Maor Ashkenazi. Performance level profiles: A formal language for describing the expected performance of functional modules. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 1751–1756. IEEE, 2016.

[6] Ronen I Brafman, Guy Shani, and Eyal Shimony. Performance level profiles. In *ICAPS'14 Workshop on Planning and Robotics (PlanRob)*, page 99–105. ICAPS, 2014.

[7] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[8] Behrmann, Gerd, Alexandre David, and Kim G. Larsen. A tutorial on uppaal 4.0 (2006). http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf, 2006.

[9] Department of Computer Science at Aalborg University (AAL) Department of Information Technology at Uppsala University (UPP). Semantics of the requirement specification language. http://www.it.uu.se/research/group/darts/uppaal/help.php?file=RSL-Semantics.shtml, 2012.

[10] Sara Fleury, Matthieu Herrb, and Raja Chatila. Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture. In *In International Conference on Intelligent Robots and Systems*, pages 842–848, 1997.

[11] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[12] Michel Ingham, Robert Ragno, and Brian C Williams. A reactive model-based programming language for robotic space explorers. *Proceedings of ISAIRAS-01*, 2001.

[13] François Félix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert. Prs: A high level supervision and control language for autonomous mobile robots. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 43–49. IEEE, 1996.

[14] Marta Kwiatkowska, Gethin Norman, and Jeremy Sproston. Probabilistic model checking of deadline properties in the ieee 1394 firewire root contention protocol. *Formal Aspects of Computing*, 14(3):295–318, 2003.

[15] Marta Kwiatkowska, Gethin Norman, Jeremy Sproston, and Fuzhi Wang. Symbolic model checking for probabilistic timed automata. *Information and Computation*, 205(7):1027–1077, 2007.

[16] Marta Z Kwiatkowska, Gethin Norman, and David Parker. Stochastic games for verification of probabilistic timed automata. In *FORMATS*, volume 9, pages 212–227. Springer, 2009.

[17] Tazeen Mahtab. *Automated verification of model-based programs under uncertainty*. PhD thesis, Massachusetts Institute of Technology, 2004.

[18] Gethin Norman, David Parker, and Jeremy Sproston. Model checking for probabilistic timed automata. *Formal Methods in System Design*, 43(2):164–190, 2013.

[19] Department of Information Technology at Uppsala University (UPP), Department of Computer Science at Aalborg University (AAL). Uppaal introduction. http://www.uppaal.org/about.shtml#introduction, 2015.

[20] Anthony Mallet Sara Fleury, Matthieu Herrb. Genom. https://www.openrobots.org/wiki/genom, 2012.

[21] Dana S Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group, 1971.

[22] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.

[23] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 3, pages 1931–1937. IEEE, 1998.

[24] Reid Simmons, Charles Pecheur, and Grama Srinivasan. Towards automatic verification of autonomous systems. In *Intelligent Robots and Systems, 2000.(IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, volume 2, pages 1410–1415. IEEE, 2000.

[25] Stavros Tripakis. *The analysis of timed systems in practice*. PhD thesis, Université Joseph Fourier, Grenoble, 1998.

[26] Wayne Wobcke. *An Operational Semantics for a PRS-Like Agent Architecture*, pages 569–580. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[27] Wayne Wobcke, Marc Chee, and Krystian Ji. Model checking for prs-like agents. *AI 2005: Advances in Artificial Intelligence*, pages 17–28, 2005.

CHAPTER A1

Concurrent module functions pseudocode

```
void concurrent_init( concurrent_info ){
    concurrent_info.concurrent_requests[ 0 ].is_active  = 0
    ...
    concurrent_info.concurrent_requests[ REQUETS_LAST_ID ].is_active  =
        0


    concurrent_info.concurrent_requests[ 0 ].satisfied  = 1
    ...
    concurrent_info.concurrent_requests[ REQUETS_LAST_ID ].satisfied  =
        1


    concurrent_info.is_process_would_like_to_access[ 0 ] = 0
    ...
    concurrent_info.is_process_would_like_to_access[ PROCESSES_LAST_ID
       ] = 0
}


void concurrent_satisfied_update_request_single( concurrent_info,
   variable_id, request_index ){
    if ( ( variable_id == concurrent_info.concurrent_requests[
       request_index].variable_id ) &&
         ( 1 == concurrent_info.concurrent_requests[request_index].
            is_active             ) ){

        if ( 1 == concurrent_info.concurrent_requests[request_index].
           is_single_range )
        {
            if ( 1 == concurrent_info.concurrent_requests[request_index
               ].satisfied )
            {
                if ( ( concurrent_info.concurrent_requests[
```

```
                    request_index].bound_lower <=
                        concurrent_info.concurrent_data[variable_id].
                            value ) &&
                         ( concurrent_info.concurrent_data[variable_id].
                             value <=
                            concurrent_info.concurrent_requests[
                                request_index].bound_upper ) )
                {
                    concurrent_info.concurrent_requests[request_index].
                        satisfied = 1
                }
                else
                {
                    concurrent_info.concurrent_requests[request_index].
                        satisfied = 0
                }

            }
        }
        else
        {
            if ( 1 == concurrent_info.concurrent_requests[request_index
                ].satisfied )
            {
                if ( ( concurrent_info.concurrent_data[variable_id].
                    value <=
                        concurrent_info.concurrent_requests[
                            request_index].bound_lower ) ||
                         ( concurrent_info.concurrent_requests[
                             request_index].bound_upper <=
                            concurrent_info.concurrent_data[variable_id].
                                value ) )
                {
                    concurrent_info.concurrent_requests[request_index].
                        satisfied = 1
                }
                else
                {
                    concurrent_info.concurrent_requests[request_index].
                        satisfied = 0
                }
            }
        }
    }

}


void concurrent_satisfied_update_request_all( concurrent_info,
```

```
    variable_id ){
    concurrent_satisfied_update_request_single( concurrent_info,
        variable_id, 0 )
    ...
    concurrent_satisfied_update_request_single( concurrent_info,
        variable_id, REQUETS_LAST_ID )
}



int concurrent_request_add( concurrent_info_type &concurrent_info,
    variable_id, request ){
    concurrent_info.concurrent_requests[variable_id]             =
        request
    concurrent_info.concurrent_requests[variable_id].satisfied   = 1
    concurrent_info.concurrent_requests[variable_id].is_active   = 1
    concurrent_info.concurrent_requests[variable_id].variable_id =
        variable_id


    concurrent_satisfied_update_request_single( concurrent_info,
        variable_id, variable_id )


    return variable_id
}


int concurrent_request_remove_request( concurrent_info, request_index )
    {
    if ( 1 == concurrent_info.concurrent_requests[request_index].
        is_active ){
        concurrent_info.concurrent_requests[request_index].is_active =
            0
    }


    return concurrent_info.concurrent_requests[request_index].satisfied
}


int concurrent_is_satisfied( concurrent_info, request_index ){
    return concurrent_info.concurrent_requests[request_index].satisfied
}


int concurrent_read( concurrent_info, variable_id ){
    return concurrent_info.concurrent_data[ variable_id ].value
```

```
}


void concurrent_write( concurrent_info, variable_id, value ){
    concurrent_info.concurrent_data[ variable_id ].value = value

    concurrent_satisfied_update_request_all( concurrent_info,
        variable_id )
}


void concurrent_signal_access(  concurrent_info, process_id, value ){
    concurrent_info.is_process_would_like_to_access[ process_id ] =
        value
}
```

CHAPTER A2

Software

A2.1 configurations.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">


<xs:element name="configurations">
   <xs:complexType>
    <xs:sequence>


      <xs:element name="setting" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" use="required"/>
          <xs:attribute name="value" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>

      <xs:element name="variable" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" use="required"/>
          <xs:attribute name="value" type="xs:string" use="required" />
          <xs:attribute name="min_value" type="xs:string" use="optional"/>
          <xs:attribute name="max_value" type="xs:string" use="optional"/>
          <xs:attribute name="is_exclusive_access" type="xs:boolean" use="optional"
              />
        </xs:complexType>
      </xs:element>

      <xs:element name="parameter" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="plp_name" type="xs:string" use="required"/>
```

```
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="value" type="xs:string" use="required" />
        <xs:attribute name="min_value" type="xs:string" use="optional"/>
        <xs:attribute name="max_value" type="xs:string" use="optional"/>
        <xs:attribute name="is_exclusive_access" type="xs:boolean" use="optional"
            />
      </xs:complexType>
    </xs:element>


  </xs:sequence>
 </xs:complexType>
</xs:element>


</xs:schema>
```

A2.2 control_graph.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">




 <xs:complexType name="conditions_type">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
   <xs:choice minOccurs="0" maxOccurs="1">
    <xs:element name="predicate_condition" type="predicate_type" />
    <xs:element name="formula_condition" type="formula_condition_type" />
    <xs:element name="not_condition" type="not_condition_type" />
    <xs:element name="forall_condition" type="forall_condition_type" />
    <xs:element name="exists_condition" type="exists_condition_type" />
    <xs:element name="AND" type="and_or_condition_type" />
    <xs:element name="OR" type="and_or_condition_type" />
   </xs:choice>
  </xs:sequence>
 </xs:complexType>


 <xs:complexType name="and_or_condition_type">
  <xs:sequence minOccurs="2" maxOccurs="unbounded">
   <xs:choice minOccurs="1" maxOccurs="1">
```

```
      <xs:element name="predicate_condition" type="predicate_type" />
      <xs:element name="formula_condition" type="formula_condition_type" />
      <xs:element name="not_condition" type="not_condition_type" />
      <xs:element name="forall_condition" type="forall_condition_type" />
      <xs:element name="exists_condition" type="exists_condition_type" />
      <xs:element name="AND" type="and_or_condition_type" />
      <xs:element name="OR" type="and_or_condition_type" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="predicate_type">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element name="field" type="field_type" />
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required" />
</xs:complexType>


<xs:complexType name="formula_condition_type">
  <xs:sequence minOccurs="1" maxOccurs="1">
    <xs:element name="expression" type="simple_value_type" />
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:sequence minOccurs="1" maxOccurs="1">
        <xs:element name="operator">
          <xs:complexType>
            <xs:attribute name="type" type="operator_type" />
          </xs:complexType>
        </xs:element>
        <xs:element name="expression" type="simple_value_type" />
      </xs:sequence>
      <xs:sequence minOccurs="1" maxOccurs="1">
        <xs:element name="inside_range">
          <xs:complexType>
            <xs:sequence minOccurs="1" maxOccurs="1">
              <xs:element name="range" type="range_type" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:choice>
  </xs:sequence>
```

```
  <xs:attribute name="key_description" type="xs:string" use="required" />
</xs:complexType>


<xs:complexType name="simple_value_type">
  <xs:attribute name="value" type="xs:string" use="required" />
</xs:complexType>


<xs:simpleType name="operator_type">
  <xs:restriction base="xs:string">
    <xs:enumeration value="=" />
    <xs:enumeration value="!=" />
    <xs:enumeration value="less" />
    <xs:enumeration value="less_equal" />
    <xs:enumeration value="greater" />
    <xs:enumeration value="greater_equal" />
  </xs:restriction>
</xs:simpleType>


<xs:complexType name="not_condition_type">
  <xs:choice minOccurs="1" maxOccurs="1">
    <xs:element name="predicate_condition" type="predicate_type" />
    <xs:element name="forall_condition" type="forall_condition_type" />
    <xs:element name="exists_condition" type="exists_condition_type" />
    <xs:element name="AND" type="and_or_condition_type" />
    <xs:element name="OR" type="and_or_condition_type" />
  </xs:choice>
</xs:complexType>


<xs:complexType name="forall_condition_type">
  <xs:sequence minOccurs="1" maxOccurs="1">
    <xs:sequence minOccurs="1" maxOccurs="unbounded">
      <xs:element name="param">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element name="predicate_condition" type="predicate_type" />
```

```
      <xs:element name="formula_condition" type="formula_condition_type" />
      <xs:element name="exists_condition" type="exists_condition_type" />
      <xs:element name="not_condition" type="not_condition_type" />
      <xs:element name="AND" type="and_or_condition_type" />
      <xs:element name="OR" type="and_or_condition_type" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="exists_condition_type">
  <xs:sequence minOccurs="1" maxOccurs="1">
    <xs:sequence minOccurs="1" maxOccurs="unbounded">
      <xs:element name="param">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element name="predicate_condition" type="predicate_type" />
      <xs:element name="formula_condition" type="formula_condition_type" />
      <xs:element name="forall_condition" type="forall_condition_type" />
      <xs:element name="not_condition" type="not_condition_type" />
      <xs:element name="AND" type="and_or_condition_type" />
      <xs:element name="OR" type="and_or_condition_type" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="field_type">
  <xs:attribute name="value" type="xs:string" use="required" />
</xs:complexType>


<xs:complexType name="range_type">
  <xs:attribute name="min_value" type="xs:string" use="required" />
  <xs:attribute name="min_inclusive" type="xs:boolean" use="required" />
  <xs:attribute name="max_value" type="xs:string" use="required" />
  <xs:attribute name="max_inclusive" type="xs:boolean" use="required" />
</xs:complexType>
```

```
<xs:simpleType name="type_start_policy">
  <xs:restriction base="xs:string">
    <xs:enumeration value="all_predecessor_done" />
    <xs:enumeration value="any_predecessor_done" />
  </xs:restriction>
</xs:simpleType>

<xs:element name="control_graph">
  <xs:complexType>
    <xs:sequence>

      <xs:element name="root">
        <xs:complexType>
          <xs:attribute name="root_name" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:choice minOccurs="0" maxOccurs="unbounded">

        <xs:element name="node_probability">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="probability_for_successor_node" minOccurs="0"
                  maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="update" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="formula_condition" type="
                              formula_condition_type" minOccurs="1" maxOccurs="1" />
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
```

```
            <xs:attribute name="probability" type="xs:string" use="required"/
                >
            <xs:attribute name="node_name" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="node_name" type="xs:ID" use="required"/>
      <xs:attribute name="start_policy" type="type_start_policy" use="
          required"/>
      <xs:attribute name="wait_time" type="xs:string" use="optional"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="node_concurrent">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="update" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="formula_condition" type="
                  formula_condition_type" minOccurs="1" maxOccurs="1" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="run_node" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:attribute name="node_name" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="node_name" type="xs:ID" use="required"/>
      <xs:attribute name="start_policy" type="type_start_policy" use="
          required"/>
      <xs:attribute name="wait_time" type="xs:string" use="optional"/>
    </xs:complexType>
  </xs:element>


  <xs:element name="node_sequential">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="run_plp" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
```

```
        <xs:sequence>
          <xs:element name="update" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="formula_condition" type="
                    formula_condition_type" minOccurs="1" maxOccurs="1" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="plp_name" type="xs:string" use="required"/>
        <xs:attribute name="wait_time" type="xs:string" use="optional"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="node_name" type="xs:ID" use="required"/>
  <xs:attribute name="start_policy" type="type_start_policy" use="
      required"/>
  <xs:attribute name="next_node_name" type="xs:string" use="optional"/>
 </xs:complexType>
</xs:element>


<xs:element name="node_condition">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="run_node" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="preconditions" type="conditions_type"
          minOccurs="1" maxOccurs="1" />
      <xs:element name="update" minOccurs="0" maxOccurs="unbounded">
       <xs:complexType>
        <xs:sequence>
         <xs:element name="formula_condition" type="
             formula_condition_type" minOccurs="1" maxOccurs="1" />
        </xs:sequence>
       </xs:complexType>
      </xs:element>
     </xs:sequence>
     <xs:attribute name="node_name" type="xs:string" use="required"/>
    </xs:complexType>
   </xs:element>
```

```
            </xs:sequence>
            <xs:attribute name="node_name" type="xs:ID" use="required"/>
            <xs:attribute name="start_policy" type="type_start_policy" use="
                required"/>
            <xs:attribute name="wait_time" type="xs:string" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>


</xs:schema>
```

CHAPTER A3

Example 2

A3.1 control_graph.xml

```
<?xml version="1.0" encoding="UTF-8"?>


<control_graph
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="control_graph.xsd">


<root root_name="node_sequential_start"/>


<node_sequential node_name="node_sequential_start" start_policy="
    any_predecessor_done" next_node_name="node_concurrent_hold_and_move">
    <run_plp plp_name="achieve_move_to">
        <update>
          <formula_condition key_description="update variables">
            <expression value="_achieve_move_to_destination" />
            <operator type="="/>
            <expression value="at_a_key" />
          </formula_condition>
        </update>
    </run_plp>
    <run_plp plp_name="achieve_key_take">
    </run_plp>
</node_sequential>


<node_concurrent node_name="node_concurrent_hold_and_move" start_policy="
    any_predecessor_done">
    <run_node node_name="node_sequential_key_holding"/>
```

```
    <run_node node_name="node_sequential_move_to_door"/>
</node_concurrent>


<node_sequential node_name="node_sequential_key_holding" start_policy="
    any_predecessor_done" next_node_name="">
    <run_plp plp_name="maintain_key_hold">
    </run_plp>
</node_sequential>


<node_sequential node_name="node_sequential_move_to_door" start_policy="
    any_predecessor_done" next_node_name="node_condition_decide_after_is_open">
    <run_plp plp_name="achieve_move_to">
        <update>
          <formula_condition key_description="update variables">
            <expression value="_achieve_move_to_destination" />
            <operator type="="/>
            <expression value="at_a_doorway" />
          </formula_condition>
        </update>
    </run_plp>
    <run_plp plp_name="observe_is_door_open">
    </run_plp>
</node_sequential>


<node_condition node_name="node_condition_decide_after_is_open" start_policy="
    any_predecessor_done">
    <run_node node_name="node_sequential_move_to_target">
        <preconditions>
            <formula_condition key_description="true == door_is_open">
              <expression value="_observe_is_door_open_door_is_open" />
              <operator type="="/>
              <expression value="TRUE" />
            </formula_condition>
        </preconditions>
    </run_node>
    <run_node node_name="node_sequential_is_door_locked">
        <preconditions>
            <formula_condition key_description="false == door_is_open">
              <expression value="_observe_is_door_open_door_is_open" />
              <operator type="="/>
```

```
            <expression value="FALSE" />
          </formula_condition>
        </preconditions>
    </run_node>
</node_condition>


<node_sequential node_name="node_sequential_is_door_locked" start_policy="
    any_predecessor_done" next_node_name="node_condition_decide_after_is_locked">
    <run_plp plp_name="observe_is_door_locked">
    </run_plp>
</node_sequential>


<node_condition node_name="node_condition_decide_after_is_locked" start_policy="
    any_predecessor_done">
    <run_node node_name="node_sequential_door_unlock">
        <preconditions>
            <formula_condition key_description="true == door_is_locked">
              <expression value="_observe_is_door_locked_door_is_locked" />
              <operator type="="/>
              <expression value="TRUE" />
            </formula_condition>
        </preconditions>
    </run_node>
    <run_node node_name="node_sequential_door_open">
        <preconditions>
            <formula_condition key_description="false == door_is_locked">
              <expression value="_observe_is_door_locked_door_is_locked" />
              <operator type="="/>
              <expression value="FALSE" />
            </formula_condition>
        </preconditions>
    </run_node>
</node_condition>


<node_sequential node_name="node_sequential_door_unlock" start_policy="
    any_predecessor_done" next_node_name="node_sequential_door_open">
    <run_plp plp_name="achieve_door_unlock">
    </run_plp>
</node_sequential>
```

```
<node_sequential node_name="node_sequential_door_open" start_policy="
    any_predecessor_done" next_node_name="node_sequential_move_to_target">
    <run_plp plp_name="achieve_door_open">
    </run_plp>
</node_sequential>




<node_sequential node_name="node_sequential_move_to_target" start_policy="
    any_predecessor_done" next_node_name="">
    <run_plp plp_name="achieve_move_to">
        <update>
          <formula_condition key_description="move through doorway">
            <expression value="_achieve_move_to_destination" />
            <operator type="="/>
            <expression value="at_b_doorway" />
          </formula_condition>
        </update>
    </run_plp>
    <run_plp plp_name="achieve_move_to">
        <update>
          <formula_condition key_description="move through doorway">
            <expression value="_achieve_move_to_destination" />
            <operator type="="/>
            <expression value="at_b_target" />
          </formula_condition>
        </update>
    </run_plp>
</node_sequential>


</control_graph>
```

A3.2 configurations.xml

```
<?xml version="1.0" encoding="UTF-8"?>


<configurations
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="configurations.xsd">
```

```
<setting name="precision_multiplier_for_numbers_and_time" value="100"/>
<setting name="observe_variable_samples" value="10"/>
<setting name="run_time_amount_of_intervals_for_discretization" value="5"/>


<variable name="robot_location" value="at_a" is_exclusive_access="true" />
<variable name="key_location" value="at_a_key" is_exclusive_access="true" />
<variable name="holding_key" value="FALSE" is_exclusive_access="true" />
<variable name="key_dropped" value="FALSE" is_exclusive_access="true" />
<variable name="key_can_drop" value="FALSE" is_exclusive_access="true" />
<variable name="door_was_opened" value="FALSE" is_exclusive_access="true" />
<variable name="door_was_unlocked" value="FALSE" is_exclusive_access="true" />


<parameter plp_name="observe_is_door_locked" name="door_is_locked" min_value="TRUE
    " max_value="FALSE" />
<parameter plp_name="observe_is_door_open" name="door_is_open" min_value="TRUE"
    max_value="FALSE" />




</configurations>
```

A3.3 achieve_door_open.xml

```
<?xml version="1.0" encoding="utf-8"?>
<plps:achieve_plp name="achieve_door_open" version="1.0" glue_file_location=""
       xmlns:plps="PLP-schemas"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="PLP-schemas AchievePLP_schema.xsd">


  <parameters>
    <execution_parameters>
    </execution_parameters>


    <input_parameters>
    </input_parameters>
```

```
  <output_parameters>
  </output_parameters>


  <non_observable>
  </non_observable>

</parameters>


<variables>
  <var name="robot_location" type="integer"/>
  <var name="door_was_opened" type="boolean"/>
</variables>


<constants>
</constants>


<required_resources>
</required_resources>


<preconditions>
  <AND>
      <OR>
          <formula_condition key_description="near the door at a">
            <expression value="robot_location"/>
            <operator type="="/>
            <expression value="at_a_doorway"/>
          </formula_condition>

          <formula_condition key_description="near the door at b">
            <expression value="robot_location"/>
            <operator type="="/>
            <expression value="at_b_doorway"/>
          </formula_condition>
      </OR>

      <OR>
          <formula_condition key_description="door_was_unlocked">
```

```
            <expression value="door_was_unlocked"/>
            <operator type="="/>
            <expression value="TRUE"/>
          </formula_condition>

          <formula_condition key_description="observed that door is unlocked">
            <expression value="_observe_is_door_locked_door_is_locked"/>
            <operator type="="/>
            <expression value="FALSE"/>
          </formula_condition>
      </OR>
  </AND>

</preconditions>


<concurrency_conditions>
</concurrency_conditions>


<concurrent_modules>
</concurrent_modules>


<side_effects>
</side_effects>


<progress_measures>
</progress_measures>


<achievement_goal>
  <formula_condition key_description="door_was_opened">
    <expression value="door_was_opened"/>
    <operator type="="/>
    <expression value="TRUE"/>
  </formula_condition>
</achievement_goal>


<success_probability>
</success_probability>
```

```
  <runtime_given_success>
    <distribution>
      <uniform>
        <lower_bound value="1" />
        <upper_bound value="2" />
      </uniform>
    </distribution>
  </runtime_given_success>


  <failure_modes>
  </failure_modes>

  <runtime_given_failure>
  </runtime_given_failure>

</plps:achieve_plp>
```

A3.4 achieve_door_unlock.xml

```
<?xml version="1.0" encoding="utf-8"?>
<plps:achieve_plp name="achieve_door_unlock" version="1.0" glue_file_location=""
       xmlns:plps="PLP-schemas"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="PLP-schemas AchievePLP_schema.xsd">


  <parameters>
    <execution_parameters>
    </execution_parameters>


    <input_parameters>
    </input_parameters>


    <output_parameters>
    </output_parameters>


    <non_observable>
```

```
  </non_observable>

</parameters>


<variables>
  <var name="robot_location" type="integer"/>
  <var name="door_was_unlocked" type="boolean"/>
</variables>


<constants>
</constants>


<required_resources>
</required_resources>


<preconditions>
  <OR>
      <formula_condition key_description="near the door at a">
        <expression value="robot_location"/>
        <operator type="="/>
        <expression value="at_a_doorway"/>
      </formula_condition>

      <formula_condition key_description="near the door at b">
        <expression value="robot_location"/>
        <operator type="="/>
        <expression value="at_b_doorway"/>
      </formula_condition>
  </OR>
</preconditions>


<concurrency_conditions>
  <formula_condition key_description="holding_key">
    <expression value="holding_key" />
    <operator type="=" />
    <expression value="TRUE" />
  </formula_condition>
</concurrency_conditions>
```

```
<concurrent_modules>
  <module name="maintain_key_hold" concurrency_type="parallel" />
</concurrent_modules>


<side_effects>
</side_effects>


<progress_measures>
</progress_measures>


<achievement_goal>
  <formula_condition key_description="door_was_unlocked">
    <expression value="door_was_unlocked"/>
    <operator type="="/>
    <expression value="TRUE"/>
  </formula_condition>
</achievement_goal>


<success_probability>
</success_probability>


<runtime_given_success>
  <distribution>
    <uniform>
      <lower_bound value="1" />
      <upper_bound value="2" />
    </uniform>
  </distribution>
</runtime_given_success>


<failure_modes>
</failure_modes>

<runtime_given_failure>
</runtime_given_failure>
```

```
</plps:achieve_plp>
```

A3.5 achieve_key_take.xml

```
<?xml version="1.0" encoding="utf-8"?>
<plps:achieve_plp name="achieve_key_take" version="1.0" glue_file_location=""
       xmlns:plps="PLP-schemas"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="PLP-schemas AchievePLP_schema.xsd">


  <parameters>
    <execution_parameters>
    </execution_parameters>


    <input_parameters>
    </input_parameters>


    <output_parameters>
    </output_parameters>


    <non_observable>
    </non_observable>

  </parameters>


  <variables>
    <var name="holding_key" type="boolean"/>
    <var name="key_location" type="integer"/>
    <var name="robot_location" type="integer"/>
  </variables>


  <constants>
  </constants>


  <required_resources>
```

```
</required_resources>


<preconditions>
  <formula_condition key_description="Robot at the location of the key">
    <expression value="key_location"/>
    <operator type="="/>
    <expression value="robot_location"/>
  </formula_condition>
</preconditions>


<concurrency_conditions>
</concurrency_conditions>


<concurrent_modules>
</concurrent_modules>


<side_effects>
</side_effects>


<progress_measures>
</progress_measures>


<achievement_goal>
  <AND>
      <formula_condition key_description="Robot holding the key">
        <expression value="holding_key"/>
        <operator type="="/>
        <expression value="TRUE"/>
      </formula_condition>
      <formula_condition key_description="Robot holding the key">
        <expression value="key_location"/>
        <operator type="="/>
        <expression value="at_unknown"/>
      </formula_condition>
  </AND>
</achievement_goal>
```

```
  <success_probability>
  </success_probability>


  <runtime_given_success>
    <distribution>
      <uniform>
        <lower_bound value="1" />
        <upper_bound value="2" />
      </uniform>
    </distribution>
  </runtime_given_success>


  <failure_modes>
  </failure_modes>

  <runtime_given_failure>
  </runtime_given_failure>

</plps:achieve_plp>
```

A3.6 achieve_move_to.xml

```
<?xml version="1.0" encoding="utf-8"?>
<plps:achieve_plp name="achieve_move_to" version="1.0" glue_file_location=""
       xmlns:plps="PLP-schemas"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="PLP-schemas AchievePLP_schema.xsd">


  <parameters>
    <execution_parameters>
    </execution_parameters>


    <input_parameters>
      <param name="destination" />
    </input_parameters>


    <output_parameters>
```

```
  </output_parameters>


  <non_observable>
  </non_observable>

</parameters>


<variables>
  <var name="robot_location" type="integer"/>
</variables>


<constants>
  <constant name="at_unknown" value="0"/>
  <constant name="at_a" value="1"/>
  <constant name="at_a_key" value="2"/>
  <constant name="at_a_doorway" value="3"/>
  <constant name="at_b_doorway" value="4"/>
  <constant name="at_b_target" value="5"/>
</constants>


<required_resources>
</required_resources>


<preconditions>
</preconditions>


<concurrency_conditions>
</concurrency_conditions>


<concurrent_modules>
</concurrent_modules>


<side_effects>
</side_effects>
```

```
<progress_measures>
</progress_measures>


<achievement_goal>
  <formula_condition key_description="at_destination">
    <expression value="robot_location"/>
    <operator type="="/>
    <expression value="destination"/>
  </formula_condition>
</achievement_goal>


<success_probability>
  <conditional_probability>
    <AND>
      <OR>
        <formula_condition key_description="">
          <expression value="robot_location"/>
          <operator type="="/>
          <expression value="at_a"/>
        </formula_condition>
        <formula_condition key_description="">
          <expression value="robot_location"/>
          <operator type="="/>
          <expression value="at_a_key"/>
        </formula_condition>
        <formula_condition key_description="">
          <expression value="robot_location"/>
          <operator type="="/>
          <expression value="at_a_doorway"/>
        </formula_condition>
      </OR>
      <OR>
        <formula_condition key_description="">
          <expression value="destination"/>
          <operator type="="/>
          <expression value="at_a"/>
        </formula_condition>
        <formula_condition key_description="">
          <expression value="destination"/>
          <operator type="="/>
```

```
        <expression value="at_a_key"/>
      </formula_condition>
      <formula_condition key_description="">
        <expression value="destination"/>
        <operator type="="/>
        <expression value="at_a_doorway"/>
      </formula_condition>
    </OR>
  </AND>
  <probability value="1" />
</conditional_probability>

<conditional_probability>
  <AND>
    <OR>
      <formula_condition key_description="">
        <expression value="robot_location"/>
        <operator type="="/>
        <expression value="at_a_doorway"/>
      </formula_condition>
      <formula_condition key_description="">
        <expression value="robot_location"/>
        <operator type="="/>
        <expression value="at_b_doorway"/>
      </formula_condition>
    </OR>
    <OR>
      <formula_condition key_description="">
        <expression value="destination"/>
        <operator type="="/>
        <expression value="at_a_doorway"/>
      </formula_condition>
      <formula_condition key_description="">
        <expression value="destination"/>
        <operator type="="/>
        <expression value="at_b_doorway"/>
      </formula_condition>
    </OR>
    <OR>
        <formula_condition key_description="door_was_opened">
          <expression value="door_was_opened"/>
          <operator type="="/>
          <expression value="TRUE"/>
```

```
        </formula_condition>

        <formula_condition key_description="observed that door is open">
          <expression value="_observe_is_door_open_door_is_open"/>
          <operator type="="/>
          <expression value="TRUE"/>
        </formula_condition>
      </OR>
    </AND>
    <probability value="1" />
  </conditional_probability>

  <conditional_probability>
    <AND>
      <OR>
        <formula_condition key_description="">
          <expression value="robot_location"/>
          <operator type="="/>
          <expression value="at_b_doorway"/>
        </formula_condition>
        <formula_condition key_description="">
          <expression value="robot_location"/>
          <operator type="="/>
          <expression value="at_b_target"/>
        </formula_condition>
      </OR>
      <OR>
        <formula_condition key_description="">
          <expression value="destination"/>
          <operator type="="/>
          <expression value="at_b_doorway"/>
        </formula_condition>
        <formula_condition key_description="">
          <expression value="destination"/>
          <operator type="="/>
          <expression value="at_b_target"/>
        </formula_condition>
      </OR>
    </AND>
    <probability value="1" />
  </conditional_probability>
</success_probability>
```

```
  <runtime_given_success>
    <distribution>
      <uniform>
        <lower_bound value="4" />
        <upper_bound value="6" />
      </uniform>
    </distribution>
  </runtime_given_success>


  <failure_modes>
  </failure_modes>

  <runtime_given_failure>
  </runtime_given_failure>

</plps:achieve_plp>
```

A3.7 maintain_key_hold.xml

```
<?xml version="1.1" encoding="utf-8"?>
<plps:maintain_plp name="maintain_key_hold" version="1.0" glue_file_location=""
  xmlns:plps="PLP-schemas"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="PLP-schemas MaintainPLP_schema.xsd">

  <parameters>
  </parameters>


  <variables>
    <var name="holding_key" type="boolean"/>
    <var name="key_dropped" type="boolean"/>
    <var name="key_can_drop" type="boolean"/>
  </variables>


  <constants>
  </constants>


  <required_resources>
```

```
</required_resources>


<preconditions>
  <formula_condition key_description="holding_key">
    <expression value="holding_key" />
    <operator type="=" />
    <expression value="TRUE" />
  </formula_condition>
</preconditions>


<concurrency_conditions>
</concurrency_conditions>


<concurrent_modules>
</concurrent_modules>


<side_effects>
</side_effects>


<progress_measures>
</progress_measures>


<maintained_condition>
  <formula_condition key_description="holding_key">
    <expression value="holding_key" />
    <operator type="=" />
    <expression value="TRUE" />
  </formula_condition>

  <initially_true />
</maintained_condition>


<success_termination_condition>
  <formula_condition key_description="key_can_drop">
    <expression value="key_can_drop" />
    <operator type="=" />
```

```
      <expression value="TRUE" />
    </formula_condition>
  </success_termination_condition>


  <failure_termination_conditions>
    <formula_condition key_description="key_dropped">
      <expression value="key_dropped" />
      <operator type="=" />
      <expression value="TRUE" />
    </formula_condition>
  </failure_termination_conditions>


  <success_probability>
  </success_probability>


  <runtime_given_success>
    <distribution>
        <!--exp>
            <lambda-rate value="0.025"/>
        </exp-->
        <uniform>
        <lower_bound value="25.01" />
        <upper_bound value="25.01" />
      </uniform>
    </distribution>
  </runtime_given_success>

  <failure_modes>
  </failure_modes>


  <runtime_given_failure>
  </runtime_given_failure>

</plps:maintain_plp>
```

A3.8 observe_is_door_locked.xml

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<plps:observe_plp name="observe_is_door_locked" version="1.0" glue_file_location="
    "
        xmlns:plps="PLP-schemas"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="PLP-schemas ObservePLP_schema.xsd">


  <parameters>
    <execution_parameters>
    </execution_parameters>


    <input_parameters>
    </input_parameters>


    <output_parameters>
      <param name="door_is_locked"/>
    </output_parameters>


    <non_observable>
    </non_observable>
  </parameters>


  <variables>
  </variables>


  <constants>
  </constants>


  <required_resources>
  </required_resources>


  <preconditions>
    <OR>
        <formula_condition key_description="near the door at a">
          <expression value="robot_location"/>
```

```
        <operator type="="/>
        <expression value="at_a_doorway"/>
      </formula_condition>

      <formula_condition key_description="near the door at b">
        <expression value="robot_location"/>
        <operator type="="/>
        <expression value="at_b_doorway"/>
      </formula_condition>
  </OR>
</preconditions>


<concurrency_conditions>
</concurrency_conditions>


<concurrent_modules>
</concurrent_modules>


<side_effects>
</side_effects>


<progress_measures>
</progress_measures>

<observation_goal_parameter>
  <param name="door_is_locked"/>
</observation_goal_parameter>

<failure_to_observe_probability>
</failure_to_observe_probability>

<failure_termination_condition>
</failure_termination_condition>

<correct_param_observation_probability>
  <probability_given_observed_value>
    <probability value="1" />
  </probability_given_observed_value>
</correct_param_observation_probability>
```

```
  <runtime_given_success>
    <distribution>
      <uniform>
        <lower_bound value="3" />
        <upper_bound value="7" />
      </uniform>
    </distribution>
  </runtime_given_success>


  <runtime_given_failure>
  </runtime_given_failure>


</plps:observe_plp>
```

A3.9 observe_is_door_open.xml

```
<?xml version="1.0" encoding="utf-8"?>


<plps:observe_plp name="observe_is_door_open" version="1.0" glue_file_location=""
       xmlns:plps="PLP-schemas"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="PLP-schemas ObservePLP_schema.xsd">


  <parameters>
    <execution_parameters>
    </execution_parameters>


    <input_parameters>
    </input_parameters>


    <output_parameters>
      <param name="door_is_open"/>
    </output_parameters>


    <non_observable>
```

```
  </non_observable>
</parameters>


<variables>
</variables>


<constants>
</constants>


<required_resources>
</required_resources>


<preconditions>
</preconditions>


<concurrency_conditions>
</concurrency_conditions>


<concurrent_modules>
</concurrent_modules>


<side_effects>
</side_effects>


<progress_measures>
</progress_measures>

<observation_goal_parameter>
  <param name="door_is_open"/>
</observation_goal_parameter>

<failure_to_observe_probability>
</failure_to_observe_probability>

<failure_termination_condition>
```

```
  </failure_termination_condition>

  <correct_param_observation_probability>
    <probability_given_observed_value>
      <probability value="1" />
    </probability_given_observed_value>
  </correct_param_observation_probability>

  <runtime_given_success>
    <distribution>
      <uniform>
        <lower_bound value="3" />
        <upper_bound value="10" />
      </uniform>
    </distribution>
  </runtime_given_success>


  <runtime_given_failure>
  </runtime_given_failure>


</plps:observe_plp>
```