Duration Probabilistic Automata

Danny Bøgsted Poulsen
dpouls09@student.aau.dk

Jonas van Vliet
jvanv09@student.aau.dk

January 24, 2011

Duration Probabilistic Automata (DPA) is a formalism for modelling concurrent execution of sequences of tasks. This formalism is well suited for scheduling problems such as job shop. We show how DPAs can be translated to other formalisms and how statistical modelchecking can be applied. The performance of these translations is tested using the tools Uppaal *and* PRISM*. The results show that the* Uppaal *translation is the fastest of the two. Furthermore, the* Uppaal *translation was found viable for verifying DPAs that consist of 55 sequences of tasks with 55 tasks each.*

1 Introduction

Modelchecking has been used to verify the correctness of a system for three decades. The problem that keeps it from being applicable to larger cases is the infamous state space explosion problem. To avoid this problem, hypothesis testing has been used to reason about the behaviour of a system. Hypothesis testing does not provide an absolute guarantee that a property holds - instead it bounds the probability of commiting an error.

Duration Probabilistic Automata(DPA) is a model that consists of sequences of tasks. Each task has a duration interval which specifies how long the task takes to complete. A probability mass function over the interval is used for discrete models, and a probability density function in the continuous case. This model is useful for modelling thread scheduling and other scheduling related problems.

Probabilistic models are convenient for expressing real-world problems. For instance, driving to work is a task that on average takes 20 minutes, but can take 15-25 minutes depending on the traffic conditions. The probability for each duration in the interval 15-25 is the same. Allowing a duration interval seems appropriate to model this compared to a fixed duration. The good case (15) is too optimistic and the worst case (25) is too pessimistic. Assuming that we leave 23 minutes before we have to be at work, the average case (20) cannot appropriately model the chance of coming late to work.

To model more real life behaviour, resources are used to model conflicts between different tasks. In the going to work setting, this could be the bathroom that each member must use before going to work. Assuming the bathroom cannot be shared this leads to conflicts.

What is a good schedule? This can be answered by a property such as "can I arrive at work before 8 in the morning with a probability greater than 80 %?". To verify such properties, tools are needed. In this article we translate the DPA model into Timed Automata and discrete time Markov Chains. We use the tools Uppaal [4] and PRISM [3] to verify properties of the DPAs on the translations. Uppaal supports probability approximation through simulation and property estimation through hypothesis testing. PRISM supports probability approximation through simulation as well as property verification. The experiments conducted concern the performance of probability approximation and property estimation. We also compare against property verification for completeness.

The rest of this article is organised as follows: Section 2 describes the DPA formalism and considers both the discrete and continuous cases. Section 3 describes hypothesis testing and an adaption of a subset of PCTL. Section 4 describes how a DPA can be translated into other formalisms by describing translations to Uppaal and PRISM. Section 5 contains experiments on the performance of the translations.

2 Duration Probabilistic Automata

In this section we introduce the formalism Duration Probabilistic Automata. This formalism was originally developed by Maler et al. [5] with inspiration from scheduling problems.

Duration Probabilistic Automata consists of tasks. A task is an abstraction of work that must be performed. Its duration is bounded by an interval $[a, b]$ and distributed according to a probability mass function φ. Consider the tasks in the driving to work example. Getting dressed beforehand takes 5-10 minutes and eating breakfast takes 3-15 minutes.

Definition 1 (Task)
A task is a triple (a, b, φ) where

- $a, b \in \mathbb{Z}_{\geq 0}$ and $a \leq b$, and
- φ is a probability mass function with range $[a, b] \cap \mathbb{Z}_{\geq 0}$. ♢

Consider having a processing unit capable of processing a single task at a time in a prescribed order. This is what we call a Simple Duration Probabilistic Automaton (SDPA). In our example, the person must first get dressed, then eat breakfast and finally drive to work.

Definition 2 (SDPA)
A Simple Duration Probabilistic Automaton is a tuple $(T, \mathcal{S}, t_1)$ where

- T is a set of tasks,
- $t_1 \in T$ is the initial task,
- $\mathcal{S} : T \to (T \cup \{\_\}) \setminus \{t_1\}$ is a one-to-one mapping that given a task returns its successor - the next task to be executed. ♢

The function $\mathcal{S}$ introduces a non-circular ordering on $T \cup \_$ ending with $\_$. The symbol $\_$ indicates that all tasks have been performed.

Initially, an SDPA must start a task. The duration of this task is drawn when starting the task with respect to its probability mass function. When the SDPA processes the task, the remaining duration decreases. The SDPA ends the task when the remaining duration reaches zero. It can then start the next task. After completing the last task, the SDPA enters a state from which it cannot move. A state of an SDPA consist of a task and the remaining duration thereof. For an SDPA $S = (T, \mathcal{S}, t_1)$ we define the set of states

$$\{(t, \_), (t, x), (\_, \_) \mid t \in T \ \wedge \ x \in \mathbb{Z}_{\geq 0}\}.$$

The semantics of S is a Markov Decision Process with initial state $(t_1, \_)$ and labels $\mathbb{Z}_{\geq 0} \cup \{start_t, end_t \mid t \in T\}$. The transition rules are shown below:

$$(start)\ \frac{}{(t, \_) \xrightarrow[\varphi(x)]{start_t} (t, x)}\ t = (a, b, \varphi), a \leq x \leq b$$

$$(delay)\ \frac{}{(t, x) \xrightarrow{d} (t, x - d)}\ d \leq x \wedge d \in \mathbb{Z}_{\geq 0}$$

$$(end)\ \frac{}{(t, 0) \xrightarrow[1]{end_t} (t', \_)}\ \mathcal{S}(t) = t', t \neq \_$$

The nondeterministic choices in this markov decision process are the delays chosen. The nondeterminism seems somewhat artificial as it is given that the SDPA from a state $(t, x), x \neq 0$ eventually ends up in a state $(t, 0)$, from where it will perform an end transition, hence the actual delays performed do not matter for the behaviour of the SDPA.

Example 3
Consider the SDPA depicted in Figure 1. Assuming the probability distribution on both tasks is uniform, the following is an execution of the SDPA:

$$\begin{aligned}(t_1^{\mathbf{1}}, \_) &\xrightarrow[\frac{1}{2}]{start_{t_1^{\mathbf{1}}}} (t_1^{\mathbf{1}}, 3) \xrightarrow{2} (t_1^{\mathbf{1}}, 1) \xrightarrow{1} (t_1^{\mathbf{1}}, 0) \\ &\xrightarrow{end_{t_1^{\mathbf{1}}}} (t_2^{\mathbf{1}}, \_) \xrightarrow[\frac{1}{3}]{start_{t_2^{\mathbf{1}}}} (t_2^{\mathbf{1}}, 7) \xrightarrow{7} (t_2^{\mathbf{1}}, 0) \\ &\xrightarrow{end_{t_2}} (\_, \_).\end{aligned}$$

*

Let $S^{\mathbf{i}} = (T^{\mathbf{i}}, \mathcal{S}^{\mathbf{i}}, t_1^{\mathbf{i}})$. We write a composition of n SDPAs as $S^{\mathbf{1}} \| S^{\mathbf{2}} \| \ldots \| S^{\mathbf{n}}$ and fix n as the number of SDPAs in a composition. In our example, this represents different people going to work. The state of a composition contains the task of each SDPA and the remaining duration of each task being processed.

Figure 1: An SDPA with two tasks.

We fix $\underline{T^{\mathbf{i}}} = T^{\mathbf{i}} \cup \{\_\}$. The task of each SDPA is then represented by a task vector in the domain $\underline{T} = \underline{T^{\mathbf{1}}} \times \underline{T^{\mathbf{2}}} \times \cdots \times \underline{T^{\mathbf{n}}}$. For a vector $\vec{t} = [t_1, t_2, \ldots, t_n]$ we write $\vec{t}[i]$ to obtain t_i and we write $\vec{t}[i/x]$ for the vector where t_i has been replaced by x.
To capture the duration of each SDPA we introduce the concept of a duration valuation. Given an index of an SDPA a duration valuation returns the remaining duration or _ if the SDPA is inactive. Formally a duration valuation for n SDPAs is a function $v : \{1, 2, \ldots, n\} \to \mathbb{Z}_{\geq 0} \cup \{\_\}$. As for task vectors let $v[i/d]$ be the valuation v' where $v'(i) = d$ and agrees with v on all other values.

The domain V consists of all possible duration valuations. The composition of SDPAs behaves as follows. Each SDPA starts its own tasks. When delaying, all SDPAs that have started a task delay simultaneously. The SDPAs that have not started a task do nothing meanwhile.

Let v be a duration valuation of a composition of n SDPAs. To find the inactive SDPAs of the composition we need only find the SDPAs with $v(i) = \_$.
For n SDPAs and a duration valuation v we define,

$$inactive(v) = \{i \mid 1 \leq i \leq n \wedge v(i) = \_\},$$

and

$$active(v) = \{1, 2, \ldots, n\} \setminus inactive(v).$$

In practice, tasks may require certain resources in a setting where only a limited amount of resources are available. A resource belongs to a specific resource type, and a task can request resources of different types. The available amount is defined by the environment. Resources can not be consumed and a task returns all resources requested when it ends. This allows resource conflicts, which are solved by introducing a scheduler.
In our example, the bathroom can be considered a limited resource. Consider a household with two bathrooms where five people live. All five people want to shower before going to work but only two of them can do so simultaneously, hence a scheduling mechanism is used to decide who can use the bathroom first.

We fix the number of resources types to be m. Then $\vec{r} \in \mathbb{Z}_{\geq 0}^m$ denotes a vector of resources such that an element of $\vec{r}$ denotes the amount available of resource number i. We denote the set of all resource vectors of size m by R. The vector $\vec{r}_{init}$ denotes the initial resources available in the environment.
We extend the notion of tasks to resource dependent tasks, which are tasks that require resources to start. The tuple representing a task is extended with a resource vector $\vec{r}$. We assume a function $req : T^{\mathbf{1}} \cup T^{\mathbf{2}} \cup \cdots \cup T^{\mathbf{n}} \to R$ that takes a task t as input and returns its resource vector. We assume that all tasks require less resources than the initial amount available in the environment.

A scheduler chooses which tasks should start and solves resource conflicts by prioritising the SDPAs.

Definition 4 (Scheduler)
A scheduler for the composition $S^{\mathbf{1}} \| S^{\mathbf{2}} \| \ldots \| S^{\mathbf{n}} \| \vec{r}_{init}$ is a function

$$sch : \underline{T} \times V \times R \to (2^{\{1,2,\ldots,n\}}, R)$$

If $sch(\vec{t}, v, \vec{r}) = (sta, \vec{u})$, then

- $\vec{u} = \vec{r} - \sum_{i \in sta} req(\vec{t}[i])$,
- for all $i \in sta : \vec{t}[i] \neq \_$ and
- $sta \cap active(v) = \emptyset$.

Furthermore, if $sch(\vec{t}, v, \vec{r}) = (\emptyset, \vec{u})$, then

- for all i, $1 \leq i \leq n$, $t_i = \_$,
- $\vec{r} \neq \vec{r}_{init}$ or
- $active(v) \neq \emptyset$. ♢

A scheduler returns the resources remaining in the environment after starting a possibly empty set of SDPAs. The scheduler never attempts to start an SDPA that has already ended or is already performing a task. When it decides not to start any tasks, it is because all SDPAs have ended, less resources are available than the total amount in the composition or some SDPA is still performing a task.

These conditions ensure progress when a composition of SDPAs is run.

The composition of n SDPAs, resources and a scheduler forms the model we work with in this paper.

Definition 5 (DPA)
A Duration Probabilistic Automaton is a composition $S^{\mathbf{1}} \| S^{\mathbf{2}} \| \dots \| S^{\mathbf{n}} \| \vec{r}_{init} \| sch$. $\diamondsuit$

For notational convenience let:

$$(v-d)(i) = \begin{cases} v(i) - d & \text{if } v(i) \neq \_ \\ \_ & \text{otherwise} \end{cases},$$

The states of a DPA $S^{\mathbf{1}} \| S^{\mathbf{2}} \| \dots S^{\mathbf{n}} \| sch \| \vec{r}_{init}$ are elements from

$$\{(\vec{t}, v, \vec{r}, s) \mid \vec{t} \in \underline{T} \wedge v \in V \wedge \vec{r} \in R \wedge s \in \{run, plan\}\}$$

A state containing *run* is essentially running, meaning it can perform delays and end tasks but not start any new tasks. States containing *plan* are planning to start tasks, and can neither delay or end tasks.

The formal transition rules for a DPA $S^{\mathbf{1}} \| S^{\mathbf{2}} \| \dots S^{\mathbf{n}} \| sch \| \vec{r}_{init}$ are shown in Figure 2. The first rule states that when the scheduler is planning, it can start SDPAs (possibly none) and then allow SDPAs to delay. The second rule performs delays in states where the SDPAs are running. This only happens when some SDPA is active. The third rule states that when the SDPAs are running and at least one SDPAs clock has reached zero, it can end and move to the next task.

Figure 3: An SDPA with two tasks.

2.1 Scheduler Specification

Different schedulers provide different sequences of task executions, hence the behaviour of a DPA is highly dependent on the scheduler applied. The scheduler used in this paper is a fixed priority scheduler (*fp*).
Consider a DPA $S^{\mathbf{1}} \| S^{\mathbf{2}} \| S^{\mathbf{3}} \| \vec{r}_{init} \| sch$, a task vector $\vec{t}$, a duration valuation v and a resource vector $\vec{r}$, such that

- $sch = fp$,
- $\vec{t} = [t^{\mathbf{1}}, t^{\mathbf{2}}, t^{\mathbf{3}}]$,
- $active(v) = \emptyset$,
- $\vec{r}_{init} = \vec{r} = [5, 2, 3]$,
- $req(t^{\mathbf{1}}) = [2, 1, 2]$, $req(t^{\mathbf{2}}) = [4, 1, 0]$, and
- $req(t^{\mathbf{3}}) = [1, 1, 1]$.

The fixed priority scheduler attempts to start the three tasks in increasing order. First it attempts to start $t^{\mathbf{1}}$. Since enough resource are available, it adds its SDPA to the list of SDPAs to start. Then it removes the resources used from the pool of available resources, hereby obtaining the resource vector $[3, 1, 1]$. It then attempts to start $t^{\mathbf{2}}$, but cannot due to a lack of resources. Finally, it attempts to start $S^{\mathbf{3}}$, which also succeeds since enough resource are available. The scheduler adds it to the list of SDPAs to start and removes the required resources from the pool of available resources and returns the list and the resources.

The implementation of the fixed priority scheduler is seen in Algorithm 1. Note that we assume the resource requirement function *req* is available in the scheduler.

2.2 Continuous DPA

A continuous time DPA is obtained by replacing the discrete random variable and probability mass function of each task with a continuous random variable and an associated probability density function. In this section we provide the intuition for understanding DPAs in a continuous time setting. A formal semantics is given by Maler et al. [5].
The probability of a continuous random variable obtaining a specific is zero thus we can no longer find the probability of a specific trace. We can however find the probability of an infinite set of traces.

Consider running the DPA in Figure 4 and assume there are no resource conflicts between any tasks.

$$start\ \frac{[(\vec{t}[i], \_) \xrightarrow[p_i]{start_{t_i}} (\vec{t}[i], x_i)]_{i \in sta}}{(\vec{t}, v, \vec{r}, plan) \xrightarrow[\Pi_{i \in sta} p_i]{start_{sta}} (\vec{t}, v[i/x_i]_{i \in sta}, \vec{u}, run)} (sta, \vec{u}) = Sch(\vec{t}, v, \vec{r})$$

$$delay\ \frac{[(\vec{t}[i], v(i)) \xrightarrow{d} (\vec{t}[i], v(i) - d)]_{i \in act}}{(\vec{t}, v, \vec{r}, run) \xrightarrow{d} (\vec{t}, v - d, \vec{r}, run)}\ \begin{array}{l} \emptyset \neq act = active(v) \text{ and} \\ d \in \mathbb{Z}_{\geq 0} \end{array}$$

$$end\ \frac{[(\vec{t}[i], v(i)) \xrightarrow{end_{t_i}} (t'_i, \_)]_{i \in e}}{(\vec{t}, v, \vec{r}, run) \xrightarrow[1]{end_e} (\vec{t}[i/t'_i]_{i \in e}, v[i/\_]_{i \in e}, \vec{u}, plan)}\ \begin{array}{l} \emptyset \neq e = \{i \mid v(i) = 0\} \text{ and} \\ \vec{u} = \vec{r} + \sum_{i \in e} req(t[i]) \end{array}$$

Figure 2: The transition rules of DPAs using a discrete semantics.

Algorithm 1: Fixed priority scheduler.

Input: A task vector $\vec{t}$, a duration valuation v, a resource vector $\vec{r}$.

Output: A set of nonnegative integers sta and a resource vector $\vec{u}$.

```
1  sta := ∅ ;
2  u⃗ := r⃗ ;
3  for i := 1 to n do
4  |  if t⃗[i] ≠ _ ∧ i ∈ inactive(v) then
5  |  |  if ∀j ∈ [1; m] : req(t⃗[i])[j] ≤ u⃗(j) then
6  |  |  |  sta := sta ∪ {i} ;
7  |  |  |  u⃗ := u⃗ − req(t⃗[i]) ;
8  |  |  end
9  |  end
10 end
11 return (sta, u⃗)
```

This means that a task is started whenever the preceding task terminates. The possible interleavings of tasks terminating are:

$$t_1^{\mathbf{1}}, t_2^{\mathbf{1}}, t_1^{\mathbf{2}}, t_2^{\mathbf{2}},$$
$$t_1^{\mathbf{1}}, t_1^{\mathbf{2}}, t_2^{\mathbf{1}}, t_2^{\mathbf{2}},$$
$$t_1^{\mathbf{1}}, t_1^{\mathbf{2}}, t_2^{\mathbf{2}}, t_2^{\mathbf{1}},$$
$$t_1^{\mathbf{2}}, t_1^{\mathbf{1}}, t_2^{\mathbf{1}}, t_2^{\mathbf{2}},$$
$$t_1^{\mathbf{2}}, t_1^{\mathbf{1}}, t_2^{\mathbf{2}}, t_2^{\mathbf{1}},$$
$$t_1^{\mathbf{2}}, t_2^{\mathbf{2}}, t_1^{\mathbf{1}}, t_2^{\mathbf{1}}.$$

We refer to these sequences as the *qualitative behaviours*[1] of the DPA. Due to timing constraints some of these are more likely than others and some might even be impossible. In the following we calculate the probability of the qualitative behaviour

$$t_1^{\mathbf{2}}, t_2^{\mathbf{2}}, t_1^{\mathbf{1}}, t_2^{\mathbf{1}}.$$

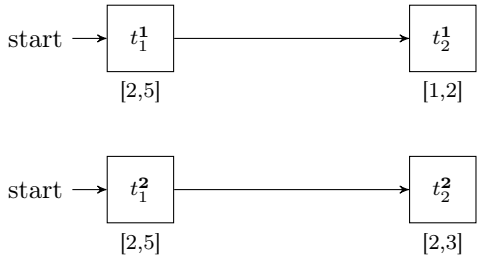


Figure 4: A DPA.

We use the convention that $\varphi_j^{\mathbf{i}}$ refers to the probability density function of $t_j^{\mathbf{i}}$ and for simplicity we let the durations be uniformly distributed. Also we write $d_j^{\mathbf{i}}$ for the duration of $t_j^{\mathbf{i}}$.

We calculate the probabilities in a stepwise fashion. For each configuration[1] the DPA is in, the probability of the correct task terminating is calculated. The probability of the qualitative behaviour is the product of all these probabilities. Initially the SDPAs processes $t_1^{\mathbf{1}}$ and $t_1^{\mathbf{2}}$. In Figure 5 the possible durations of these have been plotted as the grey area.

Since we want $t_1^{\mathbf{2}}$ to terminate before $t_1^{\mathbf{1}}$ we can deduce that $d_1^{\mathbf{2}} < d_1^{\mathbf{1}}$. On Figure 5 the area below the dashed line contains the points where $d_1^{\mathbf{2}} < d_1^{\mathbf{1}}$ thus the probability of terminating $t_1^{\mathbf{2}}$ before $t_1^{\mathbf{1}}$ is

$$\int_2^5 \int_2^{d_1^{\mathbf{1}}} \varphi_1^{\mathbf{1}}(d_1^{\mathbf{1}}) \cdot \varphi_1^{\mathbf{2}}(d_1^{\mathbf{2}}) \mathrm{d}d_1^{\mathbf{2}} \mathrm{d}d_1^{\mathbf{1}} = \frac{1}{2}.$$

While processing $t_1^{\mathbf{2}}$, time was also spent on $t_1^{\mathbf{1}}$ hence in order to terminate $t_2^{\mathbf{2}}$ before $t_1^{\mathbf{1}}$ it must be the case that

$$d_2^{\mathbf{2}} < d_1^{\mathbf{1}} - d_1^{\mathbf{2}}.$$

[1] The running tasks

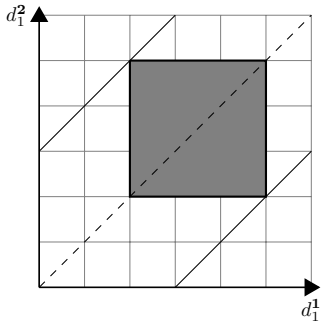


Figure 5: Plot of the possible durations of t_1^1 and t_1^2.

To find this probability we first need to know what range $d_1^1 - d_1^2$ lies in. Obviously a lower bound for this range must be when both have its smallest possible duration i.e. 2 and an upper bound must be when $d_1^2 = 2$ and $d_1^1 = 5$ thus

$$(d_1^2 - d_1^1) \in [2-2, 5-2] = [0, 3].$$

Figure 6 shows the possible remaining duration of t_1^1 and the possible duration of t_2^2.

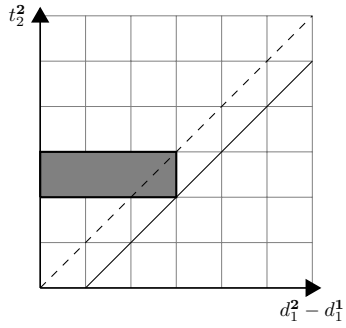


Figure 6: Possible duration for t_2^2 and the remaining duration of t_1^1 i.e. d_1^2-d_1^1

Assume we have a probability density φ over $d_1^2 - d_1^1$, then the probability of terminating d_2^2 before d_1^1 can be found by

$$\int_2^3 \int_{d_2^2}^3 \varphi_2^2(t_2^2) \cdot \varphi(t) \mathrm{d}t \mathrm{d}d_2^2,$$

where t=$d_1^2 - d_1^1$.

From probability theory a probability density is the derivative of the cumulative distribution function thus

$$\varphi(t) = P(d_1^1 - d_1^2 \leq t | d_1^2 < d_1^1)'.$$

The cumulative distribution over $d_1^2 - d_1^1$ can be found as

$$P(d_1^2 - d_1^1 \leq t \mid d_1^2 \leq d_1^1) = \frac{\int_2^{2+t} \int_2^{d_1^1} + \int_{2+t}^5 \int_{d_1^1-t}^{d_1^1}}{\int_2^5 \int_2^{d_1^1}} =$$
$$2 \cdot \left(\int_2^{2+t} \int_2^{d_1^1} + \int_{2+t}^5 \int_{d_1^1-t}^{d_1^1}\right),$$

where all integrals are over $\varphi_1^1 \cdot \varphi_1^2$ and $0 \leq t \leq 3$. Figure 7 shows how the bounds of integrals are found.

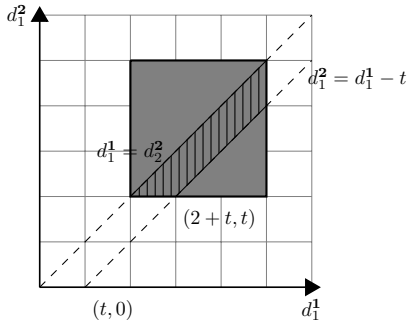


Figure 7: The points within the scratched area all have $0 \leq d_1^2 - d_1^1 \leq t$.

Using this gives

$$\int_2^3 \int_{d_2^2}^3 \varphi_2^2(d_2^2) \cdot \varphi(t) \mathrm{d}t \mathrm{d}t_2^2 = \frac{1}{27}.$$

After terminating t_2^2, the remaining tasks terminate in the order t_1^1, t_2^1 with probability 1.

The probability of the qualitative behaviour $t_1^2, t_2^2, t_1^1, t_2^1$ is, based on the above calculations,

$$\frac{1}{2} \cdot \frac{1}{27} = \frac{1}{54} \approx 1,87\%.$$

Instead of calculating the probabilities on the fly, we could instead derive the relation between the durations, for instance that $d_1^2 - d_1^1 \leq d_2^2$ and integrate over the solutions to the inequalities found. For in-

stance, the set of solutions to

$$\begin{aligned} 2 \leq d_1^{\mathbf{1}} \leq 5 \\ 2 \leq d_1^{\mathbf{2}} \leq 5 \\ 2 \leq d_2^{\mathbf{2}} \leq 3 \\ 1 \leq d_2^{\mathbf{1}} \leq 2 \\ 2 \leq d_2^{\mathbf{2}} \leq 3 \\ d_1^{\mathbf{2}} - d_1^{\mathbf{1}} \leq d_2^{\mathbf{2}} \end{aligned}$$

would constitute all the possible combinations of durations for the qualitative behaviour $t_1^{\mathbf{2}}, t_2^{\mathbf{2}}, t_1^{\mathbf{1}}, t_2^{\mathbf{1}}$. Integrating $\varphi_1^{\mathbf{1}} \cdot \varphi_2^{\mathbf{1}} \cdot \varphi_1^{\mathbf{2}} \cdot \varphi_2^{\mathbf{2}}$ over the solution space of the inequalities would provide the probability of the behaviour. This idea was proposed by Bozga et al. [1].

3 Logic specifications and statistical model checking

In this section we introduce the logic in which we specify requirements to DPAs. Furthermore we review hypothesis testing as this has applications in statistical model checking. The material covered in this section applies for both discrete and continuous DPAs unless explicitly stated otherwise in the text.

3.1 Trace

A sequence of transitions of a DPA is called a trace of the DPA. The probability of a trace from a discrete[2] DPA is defined as the product of the individual probability transitions. The composition of the SDPAs in Figure 1 and 3, and $\vec{r}_{init} \| fp$, where

- $\vec{r}_{init} = [5]$,
- $req(t_1^{\mathbf{1}}) = [3]$, $req(t_2^{\mathbf{1}}) = [5]$,
- $req(t_1^{\mathbf{2}}) = [2]$, and $req(t_2^{\mathbf{2}}) = [1]$,

form a DPA that can create the following trace.

$$\begin{aligned} &([t_1^{\mathbf{1}}, t_1^{\mathbf{2}}], [\_, \_], [5], plan) \xrightarrow[\frac{1}{2} \cdot \frac{1}{3}]{start_{\{1,2\}}} \\ &([t_1^{\mathbf{1}}, t_1^{\mathbf{2}}], [3, 1], [0], run) \xrightarrow[1]{1} \\ &([t_1^{\mathbf{1}}, t_1^{\mathbf{2}}], [2, 0], [0], run) \xrightarrow[1]{end_{\{2\}}} \\ &([t_1^{\mathbf{1}}, t_2^{\mathbf{2}}], [2, \_], [2], plan) \xrightarrow[\frac{1}{5}]{start_2} \\ &([t_1^{\mathbf{1}}, t_2^{\mathbf{2}}], [2, 5], [1], run) \xrightarrow[1]{2} \\ &([t_1^{\mathbf{1}}, t_2^{\mathbf{2}}], [0, 3], [1], run) \xrightarrow[1]{end_{\{1\}}} \\ &\ldots \end{aligned}$$

The trace has the probability $\frac{1}{2} \cdot \frac{1}{3} \cdot \frac{1}{5} \cdot \ldots$. By $Pr(\tau)$ we denote the probability of the trace τ and by $traces(s)$ we denote all traces emanating from the state s.
The path of a trace is the sequence of states that the trace passes through annotated with the global time passed. The path over the trace shown before is

$$\begin{aligned} &([t_1^{\mathbf{1}}, t_1^{\mathbf{2}}], [\_, \_], [5], plan)^0, ([t_1^{\mathbf{1}}, t_1^{\mathbf{2}}], [3, 1], [0], run)^0, \\ &([t_1^{\mathbf{1}}, t_1^{\mathbf{2}}], [2, 0], [0], run)^1, ([t_1^{\mathbf{1}}, t_2^{\mathbf{2}}], [2, \_], [2], plan)^1, \\ &([t_1^{\mathbf{1}}, t_2^{\mathbf{2}}], [2, 5], [1], run)^1, ([t_1^{\mathbf{1}}, t_2^{\mathbf{2}}], [0, 3], [1], run),^3 \\ &\ldots. \end{aligned}$$

To obtain the path of a trace τ we write $path(\tau)$.

3.2 Logic

Requirements to a DPA are specified in Probabilistic Computation Tree Logic (PCTL) developed by Hansson and Jonsson [2].

Definition 6 (PCTL)
The syntax of a state formula is generated by the abstract syntax,

$$\begin{aligned} f, f_1, f_2 ::= & tt \mid a \mid \neg f \\ & f_1 \wedge f_2 \mid \mathcal{P}_{\bowtie \theta}(\gamma) \end{aligned}$$

where a is an atomic proposition, $\bowtie \in \{\leq, \geq\}$, $\theta \in [0, 1]$ and γ is generated by the syntax for path formulae below

$$\gamma ::= \mathcal{X} f \mid f_1 \mathcal{U} f_2 \mid f_1 \mathcal{U}_{\leq d} f_2,$$

where $d \in \mathbb{R}_{\geq 0}$. ♢

[2] In the continuous time setting we find the probability of a set of traces, not a specific trace

$$
\begin{array}{ll}
s \vDash tt & \\
s \vDash a & \text{if } a \in AP(s) \\
s \vDash \neg f & \text{if } \neg(s \vDash f) \\
s \vDash f_1 \wedge f_2 & \text{if } s \vDash f_1 \wedge s \vDash f_2 \\
s \vDash \mathcal{P}_{\bowtie\theta}(\gamma) & \text{if } Pr(\{\tau \mid \tau \in traces(s) \wedge path(\tau) \vdash \gamma\}) \bowtie \theta \\
s_1^{d_1}, s_2^{d_2}, \ldots, s_n^{d_n} \vdash \mathcal{X} f & \text{if } s_2^{d_2} \vDash f \\
s_1^{d_1}, s_2^{d_2}, \ldots, s_n^{d_n} \vdash f_1 \,\mathcal{U}\, f_2 & \text{if there exists an } i \leq n \text{ s.t. } s_i^{d_i} \vDash f_2 \\
 & \text{and } \forall 1 \leq j < i, s_j^{d_j} \vDash f_1. \\
s_1^{d_1}, s_2^{d_2}, \ldots, s_n^{d_n} \vdash f_1 \,\mathcal{U}_{\leq d}\, f_2 & \text{if there exists an } i \leq n \text{ s.t. } s_i^{d_i} \vDash f_2 \\
 & \text{and } \forall 1 \leq j < i, s_j^{d_j} \vDash f_1 \\
 & \text{and } d_i - d_1 \leq d.
\end{array}
$$

Figure 8: Satisfaction relations

We define a satisfaction relation ($\vDash$) between a state formula and a state (s) and a satisfaction relation ($\vdash$) between a path and a path formula. In the satisfaction rules in Figure 8 we assume a function AP that generates all atomic propositions satisfied by a state.

The atomic propositions we consider are whether a task is active, if a task has been executed or if a task is waiting to be executed. Formally, the set of atomic proposition over a DPA $D = S^{\mathbf{1}} \| S^{\mathbf{2}} \| \ldots \| S^{\mathbf{n}}$ is the set $\{t.active, t.ended, t.waiting \mid t \in T^{\mathbf{1}} \cup T^{\mathbf{2}} \cup \cdots \cup T^{\mathbf{n}}\}$. We say a DPA D satisfies a state formula f if the initial state of D satisfies f.

Using the logic we express properties such as "*Does* t_1 *start before* t_2 *with certainty greater than* 0.2" as

$$\mathcal{P}_{\geq 0.2}(\neg t_2.active \,\mathcal{U}\, t_1.active)$$

and "*Do* t_1 *and* t_2 *run at the same time with certainty greater than* 0.1" as

$$\mathcal{P}_{\geq 0.1}(tt \,\mathcal{U}\, t_1.active \wedge t_2.active).$$

Determining whether $s \vDash \mathcal{P}_{\bowtie\theta}(\gamma)$ holds requires computing $Pr(\{\tau \mid \tau \in traces(s) \wedge path(\tau) \vdash \gamma\})$. To find the exact probability we must examine all traces and determine how large a percentage of these satisfy γ. This approach is not viable in practice in large cases, as it will encounter the state space explosion problem. In the next section we describe how the probability can be estimated and error bounds can be given.

3.3 Hypothesis testing

Verifying a property often requires exploring branches of the underlying transition system until the property is proven true or false. In many cases, all reachable states must be explored. This is a problem for large systems since the size of the state space grows exponentially with the size of the model - often called the state space explosion problem. This makes verifying these system impractical. To avoid this problem, hypothesis testing[6] estimates the probability that a property holds based on simulations of a system. This method cannot guarantee a correct result, but bounds on the probability of making an error have made this a viable approach to model checking.

Consider a DPA D and a property $\mathcal{P}_{\geq\theta}(f)$, where f does not contain any probabilistic operators. Since f does not contain probabilistic operators, we can always decide whether a trace satisfies f or not. Using this we generate a sample of traces of D and evaluate how many of these satisfy f. Let k be the number of traces satisfying f and let m be the total number of traces in the sample. Furthermore let η be the real probability of an arbitrary trace satisfying f. In term from theory of probabilities, the m traces are an experiment with the k number of traces satisfying f being the outcome thus we have of discrete random variable X over this experiment. As a given trace either satisfy f or not , X has a binomial distribution with the success parameter η. .

To verify whether D satisfies $\mathcal{P}_{\geq\theta}(f)$ we use our sample to test the hypothesis

$$H_0 : \eta \geq \theta$$

against

$$H_1 : \eta < \theta.$$

In case H_0 is a true hypothesis then the sample should reflect this thus we expect that $k \geq \theta \cdot m$. If this is indeed the case we accept H_0[3]. However, as the sample is drawn at random it may be the case that $k < \theta \cdot m$ although H_0 is correct. As an example consider the case where $\theta = 0.4$ and $m = 100$ thus we expect k to be 40. If $k = 35$ it is still reasonable to think that H_0 is true whereas if $k = 10$ it is unlikely that H_0 is correct. In the latter case, we refute H_0 in favour of H_1, although H_0 could be correct.

To specify when to refute H_0, a level of *significance* is required. The level of significance describes the willingness to reject a true H_0. Let α be the significance level, then H_0 is rejected if and only if $k < c$ where $c \leq m$ is the biggest integer such that

$$\sum_{i=0}^{c}(\binom{m}{i}\theta^i \cdot (1-\theta)^{m-i}) \leq \alpha,$$

where $\binom{m}{i}$ is the binomial coefficient. The range 0 to c is known as the critical region.

Example 7
Let $\mathcal{P}_{\geq 0.4}(f)$ be a property we wish to verify and let f have no probabilistic operators. A sample of 100 traces has been obtained of which 35 satisfy f. Let η be the real probability of a arbitrary trace satisfying f. To verify the property we test the hypothesis

$$H_0 : \eta \geq 0.4,$$

at significance level $\alpha = 0.05$.

Using the aforementioned approach we find the biggest integer between 0 and 100 for which

$$\sum_{i=0}^{c}(\binom{100}{i}0.4^i \cdot (1-0.4)^{100-i}) \leq 0.05.$$

In this case $c = 31$. As $c < 35$ we accept H_0 and thereby accept that $\mathcal{P}_{\geq 0.4}(f)$ is satisfied by the system.

[3]We cannot refute H_0, but this does not prove that it is correct.

The level of significance provides a way to bound the probability of rejecting a true hypothesis - known as a type-1 error. The second kind of error one can commit is a to accept a false hypothesis - a type-2 error. Bounding this probability is more difficult as it depend on the true value of η.
What we can do is to find a *power* function $g : [0,1] \to [0,1]$ that given a parameter produces the probability of rejecting H_0. In Example 7 the power function would for instance be

$$g(\eta) = \sum_{i=0}^{31}(\eta)^i \cdot (1-\eta)^{100-i},$$

thus the probability of obtaining an observation in the critical region given that the true probability is η.

Instead of only bounding the type-1 error we may also bound the probability of committing a type-2 error. To do so we introduce an indifference region[7]. Let $\mathcal{P}_{\geq\theta}(f)$ be the property we wish to test, then our indifference region is

$$[\theta - \delta, \theta + \delta],$$

for some δ. Let η be the actual probability of satisfying f.
The hypotheses we test are then

$$H_0 : \eta \geq \theta + \delta$$

against

$$H_1 : \eta \leq \theta - \delta.$$

Again let m be our sample size and let k be the number of traces satisfying f. Furthermore let α be our wanted level of significance and let β be the wanted probability of committing a type-2 error i.e. the power of the test. Choosing the biggest integer $c \leq m$ such that

$$\sum_{i=0}^{c}\binom{m}{i}(\theta+\delta)^i \cdot (1-(\theta+\delta))^{m-i} \leq \alpha \qquad (1)$$

and accepting H_0 if $k > c$ ensures the probability of committing a type-1 error is less than α.
In case

$$1 - \sum_{i=0}^{c}\binom{m}{i}(\theta-\delta)^i \cdot (\theta-\delta)^{m-i} \leq \beta \qquad (2)$$

we say that the test has power β. In practise it means that if $\eta \leq \theta - \delta$ then the probability of accepting H_0

is less than β.
If we can generate sample traces, it is possible to fix β and instead find values of m and c by solving (1) and (2). Multiple solutions of course exists to these equations but to minimise the verification effort we wish to find one with low m. In his thesis, Younes [7] proposes an algorithm based on binary search that given the indifference region $(\theta \pm \delta)$, α and β finds m and c.

4 Translations

In this section we translate the DPA model into other modelling formalisms. This allows us to use existing verification tools to verify DPA models. We have chosen to use the Network of Timed Automata formalism (used in the tool UPPAAL) and discrete time markov chains (used in the tool PRISM). In both cases, we present translations that are based on an assumption of a uniform distribution over the tasks durations.

4.1 PRISM

The core component of a PRISM program is a module. A module consists of variables and guarded transitions with a probabilistic outcome that alters the variables. The variables are bounded integers or booleans.

In our translation a module is created for the scheduler and a module is created for each of the SDPAs.

4.1.1 SDPA module

For each SDPA $S^{\mathbf{i}}$ we create a PRISM module that has the variables

- $clockS_i$ with range $1, 2, 3, \ldots maxC$ where $maxC$ is the maximal duration of any task of $S^{\mathbf{i}}$,
- $activeS_i$ with range $true, false$,
- $taskS_i$, with range $1, 2, 3, \ldots, |T^{\mathbf{i}}| + 1$ and
- $paybackS_i$ with range $true, false$.

The variables $clockS_i$, $activeS_i$ and $taskS_i$ are used to encode the state of the SDPA. For instance, a state $(t^{\mathbf{i}}_j, x)$ is encoded by setting $clockS_i = x$, $activeS_i = true$ and $taskS_i = j$. A state $(t^{\mathbf{i}}_j, \_)$ is encoded by setting $taskS_i = j$ and $activeS_i = false$. The state $(\_, \_)$ is similarly encoded by setting $taskS_i = |T^{\mathbf{i}}| + 1$.
When starting $S^{\mathbf{i}}$ the scheduler allocates resources for it thus when $S^{\mathbf{i}}$ terminates the scheduler must reclaim these resources. By setting $paybackS_i$ to $true$ the SDPA indicates to the scheduler that it has terminated and that the scheduler should reclaim the resources. When the resources have been reclaimed $paybackS_i$ must be reset to $false$.
The following snippet stems from an SDPA with 2 tasks where the maximal duration of any task is 7 time units.

```
module S1
taskS1 : [1..3] init 1;
clockS1 : [0..7] init 7;
paybackS1 : bool init false;
activeS1 : bool init false;
```

A statement such as taskS1 : [1..3] init 1 declares a variable taskS1 that can obtain the values $1, 2, 3$ and initially set to 1[4]. As taskS1 maximally obtains the value 3 we see this SDPA must execute 2 tasks. To start processing a task the SDPA module awaits synchronisation on its start label *startS1*. The transition rules starting the intial task are shown below. The guards of this rule states that

- S1 must not be active,
- S1 must not need to repay any resources, and
- the task to be started must be $t^{\mathbf{1}}_1$.

```
[startS1] (!activeS1 & !paybackS1
            & taskS1=1)
          ->
 1/2:(activeS1'=true) & (clockS1'=2)+
 1/2:(activeS1'=true) & (clockS1'=3);
```

The outcome of this rule is setting activeS1 to *true* and clockS1 to either 2 or 3[5]. This choice is probabilistic hence with probability $\frac{1}{2}$ $clockS_0$ is set to 2 and with probability $\frac{1}{2}$ to 3.

[4] In general variable declarations have the form: identifier : type init value where value is the initial value of the type type identified by identifier

[5] Transitions have the form [sync-label] guard -> probability:updates+probability:updates+.... The sync label is optional and used to force two modules to move simultaneously. Multiple updates are separated by & and in updates, variable assignments are denoted by "'=".

Time must start after starting t_1^1. This is accomplished by the transition rule below.

```
[delay] (activeS1 & clockS1>0 &
         taskS1<3 &
         (!paybackS1))
           ->
    1/1:(clockS1'=clockS1-1);
```

The guard of this rule states that

- the SDPA must be active,
- the clock must be greater than zero,
- the SDPA must not have finished all of its task and
- the scheduler must not need to reclaim any resources.

In case $S^{\mathbf{i}}$ is not active but another SDPA is, a *dummy* delay rule is necessary. The rule does nothing except allowing other SDPAs to delay. The rule is required because PRISM requires every module to participate in a synchronisation. This only applies if some rule of the module synchronises on the channel.

```
[delay] ((!activeS1) &
         (!paybackS1))
           ->
    1/1:(activeS1'=false);
```

During execution clockS1 eventually reaches zero and the module needs to inform the scheduler module it has finished. This happens by synchronising on the *end* label.

```
[end] (activeS1 & clockS1=0 &
       taskS1<3 &
       (!paybackS1))
         ->
  1/1:(activeS1'=false) &
      (taskS1'=taskS1+1) &
      (clockS1'=7) &
      (paybackS1'=true);
```

The guard of the rule states that

- $S^{\mathbf{1}}$ should be active,
- its clock should be zero,
- it has not processed its final task, and
- it should not need to repay resources.

If these conditions are satisfied it goes to inactive (by setting $activeS_1$ to *false*), moves on to the next task (by incrementing $taskS_1$) and sets the clock to some value greater than zero. It also indicates to the scheduler that its resources should be reclaimed (by setting $paybackS_1$ to true).

Two dummy rules are used for the end transitions in the module.

```
[end] ((!clockS1=0) & (!activeS1) &
       (!paybackS1))
         ->
  1/1:(activeS1'=false);

[end] (activeS1 & (!clockS1=0) &
   min(clockS1,clockS2)=0
    & taskS1<3 & (!paybackS1))
        ->
  1/1:(clockS1'=clockS1);
```

The first dummy rule is applied when the SDPA is not active and the second is used when the SDPA is active but has not finished its task (!clockS1=0). The min(clockS1,clockS2) is used to get the minimum value of all clocks.

The final transition rule used by the SDPA module is the payback rule. This rule synchronises with the scheduler and is used to inform the SDPA that the scheduler reclaims the resources hence the SDPA should set its payback variable to *false*.

```
[S1payback1] (paybackS1 & taskS1=2)
   -> 1/1:(paybackS1'=false);
```

Although the payback is made for t_1^1, taskS1 must be equal to two. This is because we incremented taskS1 while ending t_1^1.

4.1.2 Scheduler module

For a DPA consisting of n SDPAs and m resource types the scheduler contains the variables:

- *counter* with range 0 to $n+1$,
- *planning* with range *false* and *true*,
- *doPayback* with range *false* and *false*,
- *timePassed* with the range 0 to $maxTime$, where $maxTime$ is the time used if all tasks were processed sequentially after each other.

and an integer variable res_i for all i in $[1, m] \cap \mathbb{Z}_{\geq 0}$.

Below is an example of a scheduler for one resource type and two SDPAs ($n = 2$).

```
module scheduler
doPayback : bool init false;
counter : [1..3] init 1;
planning : bool init true;
res1 : [0..10] init 10;
timePassed : [0..120];
```

The *planning* and *doPayback* variables are used to encode the state of the scheduler. If *planning* = *true* and *doPayback* = *true* the scheduler is in a state where it reclaims resources from tasks that has terminated. If *planning* = *true* and *doPayback* = *false* the scheduler is calculating which SDPAs to start. This corresponds to *plan*-states from the DPA semantics. If *planning* = *false* the scheduler is inactive and awaits the termination of some task, corresponding to the *run*-states. From the above we see the scheduler starts in a state where it can start SDPAs.

When starting SDPAs the scheduler uses the *counter* variable to iterate through all SDPAs, evaluates if enough resources are available and if possible start them and decrease the resources variables. This corresponds to the for-loop in Algorithm 1 and *counter* corresponds to i in the algorithm.
Let t_1^1 require 7 of resource res_1 then the start rule is:

```
[startS1] ((!doPayback) & planning &
           counter=1
            & taskS1=1 & (!activeS1)
            & res1>=7)
                 ->
        1/1:(counter'=counter+1) &
            (res1'=res1-7);
```

This rule states that if the scheduler is starting SDPAs, we are considering SDPA 1, task number 1, it is not active and enough resources are available, then subtract 7 from the resource type $res1$, synchronise on the start label and consider the next SDPA.
The scheduler moves to the next SDPA if the SDPA is running, insufficient resources are available or the SDPA has finished.

```
[] (planning & counter=1 & taskS1=1
    & (!activeS1) & (res1<7)
    &  (!doPayback))
  -> 1/1:(counter'=counter+1);

[] (planning & counter=1 & taskS1=3
    & (!activeS1) & (!doPayback))
   ->
   1/1:(counter'=counter+1);

[] (planning & counter=1 & taskS1<3
    & activeS1 & (!doPayback))
   -> 1/1:(counter'=counter+1);
```

The scheduler enters a waiting state when *counter* = $n + 1$ i.e. when all SDPAs have been attempted started.

```
[](((!doPayback) & planning & counter=3)
   -> 1/1:(planning'=false);
```

The scheduler enters the payback state when some SDPA finishes its task by synchronising on the *end* label.

```
[end] ((!planning) & (!(taskS1=3
        & taskS2=3 & true)))
-> 1/1:(counter'=1) & (planning'=true)
        & (doPayback'=true);
```

In this state the scheduler works in much the same way as when starting SDPAs. It iterates through all SDPAs (using *counter*) and reclaims resources if an SDPA has finished.

```
[] (counter=1 & (doPayback) &
    (!paybackS1))
       ->
   1/1:(counter'=counter+1);

[S1payback1] (counter=1 &
     doPayback) &
      res1<=3)
       ->
  1/1:(counter'=counter+1)
      & (res1'=res1+7);
```

The first rule is applied when S^1 has no resources to repay. The second rule is used when it has. The guard res1<=3 is required by PRISM to avoid overflowing res_1.

Finally the scheduler starts new SDPAs when it has reclaimed resources from all SDPAs (hence *counter* = $n + 1$).

```
[] (counter=3 & doPayback & (!paybackS1)
    & (!paybackS2) & planning)
   -> 1/1:(counter'=1)
      & (doPayback'=false);
```

We let the scheduler synchronise on the delay label to ensure that SDPAs only delay when the scheduler is not planning,

```
[delay] ((!planning) & timePassed<120)
  -> 1/1:(planning'=false) &
        (timePassed'=timePassed+1);
```

Because PRISM requires every module that attempts to synchronise to participate in the synchronisation, this rule ensures delaying does not occur while the scheduler is planning. Additionally we use the rule to keep track of the time passed.

4.1.3 Logic translation

PRISM supports PCTL where atomic propositions are boolean expressions over the variables. Translating a logic specification for a DPA into a PRISM specification first involves translating the logic atomic propositions. Let $t_j^{\mathbf{i}}$ be the j^{th} task to be executed of SDPA $S^{\mathbf{i}}$. The atomic propositions of the DPA logic is translated as

$$\begin{aligned} t_j^{\mathbf{i}}.active &::= taskS_i = j \wedge activeS_j = true \\ t_j^{\mathbf{i}}.waiting &::= taskS_i = j \wedge activeS_j = false \\ t_j^{\mathbf{i}}.ended &::= taskS_i > j \end{aligned}$$

State formulae are easily translated into PRISM but path formulae require special attention. The cause of the problem is the scheduler encoding. Instead of starting tasks simultaneously they are started in a step-wise fashion [6]. Resources are reclaimed in a similar way. This means an end transition in the DPA semantics corresponds to a series of transitions in the PRISM translation and likewise for the start transition. See Figure 9

The intermediate states resulting from these transitions must be ignored while evaluating a formula. A path formula like

$$f_1 \mathcal{U} f_2$$

is therefore translated to

$$(f_1 \vee inter) \mathcal{U} (f_2 \wedge \neg inter),$$

where *inter* is an expression characterising the intermediate states introduced by the translation.

[6] When $planning = true$

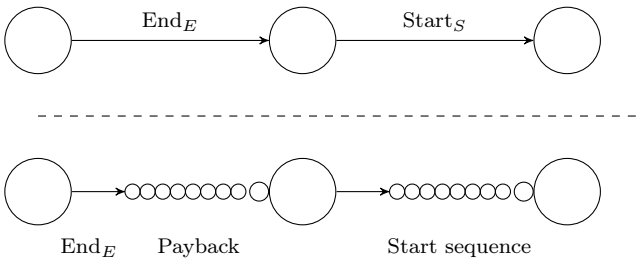


Figure 9: The top represents the states visited in the DPA semantics. The bottom represents the states visited in the PRISM translation. The small circles represent intermediate states that are not equal to any DPA state.

The intermediate states in the translation are those in which the scheduler is reclaiming resources ($planning = true$, $doPayback = true$ and $counter < n+1$) and those in which the scheduler is starting SDPAs ($planning = true$ and $doPayback = false$). The states where $planning = true$, $prismDoPayback = true$ and $counter = n+1$ correspond to *plan* states in the DPA semantics. States with $planning = false$ correspond to *run* states in the DPA semantics.
On basis of the above we conclude that the *inter* expression should characterise intermediate states as all those where the scheduler is active except for the states corresponding to *plan* states. Thereby *inter* is

$$planning = true \wedge \neg \begin{pmatrix} planning = true \wedge \\ doPayback = true \wedge \\ counter = n+1 \end{pmatrix}.$$

A similar construction, which we do not show, is needed for translating $\mathcal{X} f$.
Due to our encoding of time, we cannot translate $f_1 \mathcal{U}_{\triangleright\triangleleft d} f_2$ into PRISM.

4.2 Uppaal

Uppaal extends the Network of Timed Automata formalism with variables. These variables can be accessed directly through both guards and updates through functions written in a C-like language or variable references. It is in this setting that we translate a DPA into a Uppaal model.

4.2.1 SDPA modeling

The translation of n SDPAs creates a TA for each SDPA and a single TA modelling the scheduler. The following are local variables used by each SDPA model:

- **int id** is a unique identifier given to each SDPA.
- **int duration** is assigned a value in the duration interval of a task.
- **clock v** is the clock associated to this model.

The following are variables shared between all TAs that are used for signaling and control:

- **broadcast channel start, end** are one-to-many communication channels.
- **bool startNext[n]** is an array of flags used to signal which SDPAs must start.
- **tasksCompleted[n]** is an array containing information regarding the number of tasks each SDPA has completed.
- **ended[n]** is an array of flags used to signal which SDPAs have ended.

Figure 10 shows how an SDPA consisting of a single task with duration $[2, 3]$ is translated. The location $waitingforTask1$ represents the state where the task waits until the start signal is given. The start signal is received over the channel *start*, but in order to make the transition, the guard must be satisfied. The scheduler is responsible for both sending the start signal and setting the values in $startNext$, hereby enabling the guard. When a task is started, it immediately resets its clock to zero and sets its startNext flag to false. The model then enters a committed location, in which it cannot delay. When the model moves out of this location, it encounters a nondeterministic choice, where the task is assigned a duration. When UPPAAL simulates a TA, it chooses uniformly amongs the nondeterministic choices available. There exists an edge for each integer in the duration interval. In the following location, $task1$, the invariant and the guards on the outgoing edges ensure that the TA must wait for the amount of time previously chosen.

When enough time has passed, two edges can lead it to the location *End*. The upper edge sends a signal over the *end* channel (denoted by !), while the lower edge waits for a signal over the end channel (denoted by ?). The lower edge is used when two SDPAs end simultaneously, as only one SDPA can send a signal over the end channel. When either of the edges is taken, the SDPA registers that a task was completed and sets $ended[id] = true$ to indicate that it was one of the SDPAs ending a task.

A SDPA consisting of several tasks is modelled by replacing the *End* location with the translation of an SDPA consisting only of the next task.

4.2.2 Scheduler modeling

We consider the fixed priority scheduler. Resources are identified by their index. The scheduler contains an internal representation of the resources available and the resource requirements of each task through the following variables:

- **int resources[m]** is an array representing the resource vector of available resources.
- **int resUsage[n][k][m]** is a double nested array containing the resource requirements for each SDPA and task.
- **bool waiting[n]** is an array of flags. Each flag indicates whether the SDPA with the same index is waiting to run a task. Initialised with all flags set to true.

The values of these variables are initialised according to the model. During the execution of the model, **resUsage** never changes.

Figure 11 shows the scheduler implementation. The functions perform the following:

- **chooseStart()** This function is an implementation of Algorithm 1. To start an SDPA, it sets its flag in **startNext** to true and sets it flag in **waiting** to false.
- **release()** Releases the resources of the tasks just finished (tasks are derived from *ended* and $tasksCompleted$). Also sets the SDPAs flags in **waiting** to false. When more than one SDPA end simultaneously, the SDPAs are added to the queue in a fixed order based on the SDPA id.

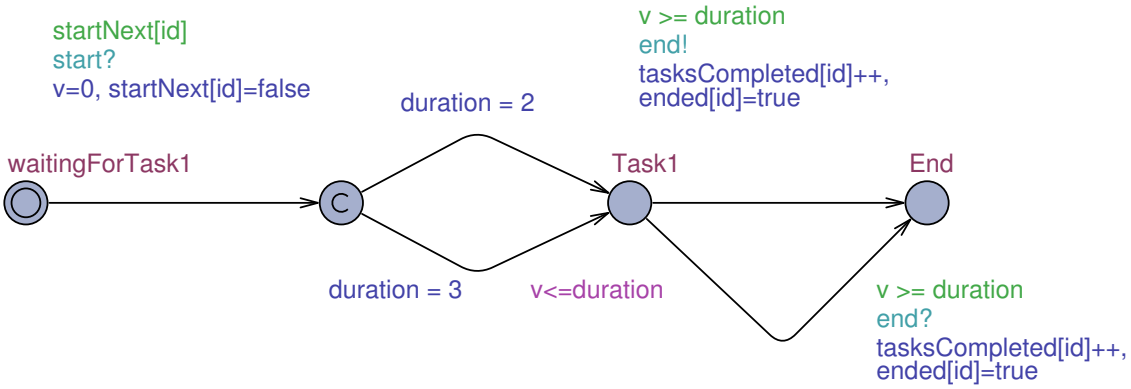


Figure 10: A task modelled in UPPAAL.

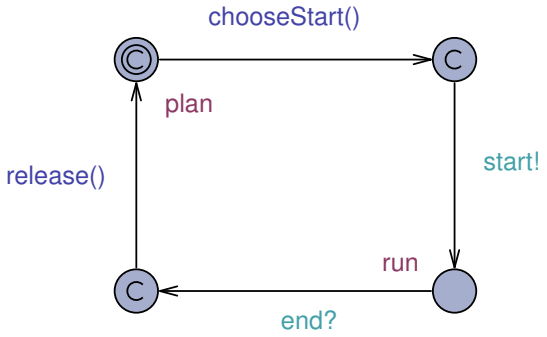


Figure 11: A *fp* scheduler modelled in UPPAAL.

The initial location *plan* only allows starting SDPAs. The edge labeled with chooseStart sets the startNext flags of the SDPAs it has determined can start. It then sends the signal to start to all SDPAs and enters the *run* location. Until this point, the scheduler has been in committed locations and time has not been able to pass. Now the scheduler must wait for an end signal by some SDPA and time passes. When this happens, it releases the resources and starts the next round of SDPAs.

4.2.3 Comparison with DPA semantics

We say the DPA and the UPPAAL model perform the same actions if:

- When a DPA starts the SDPAs in the set S, the scheduler module performs *chooseStart* and sets the *startNext* flag for all SDPAs in S, followed by a synchronisation over the *start* channel.
- When a DPA delays d time units, the UPPAAL model also delays d time units.
- When a DPA ends the SDPAs in the set E, the UPPAAL module synchronises over the channel *end* followed by the scheduler performing *release*.

From a correctness point of view we wish to show that a DPA model and its translation match each other. This is possible by proving that a bisimulation exists between the initial state of both models.

4.2.4 Logic translation

UPPAAL does not support PCTL. It is possible to translate queries on the form $\mathcal{P}_{\leq 0.8}(tt\,\mathcal{U}_{\leq 90}\, t_3^1.active \wedge t_2^3.ended)$. Such a query is translated into *P[time <= 90] (<> (SDPA1.Task3 & tasksCompleted[2] > 2)) <= 0.8.*

4.3 Continuous translation

In this section we translate the continuous time DPA to a UPPAAL model.
This UPPAAL translation is similar to the discrete time translation to UPPAAL and uses many of the same variables. The variables to model the tasks are similar, except that **broadcast channel end** has been replaced by the array of channels **broadcast channel end[n]**, so that each SDPA has its own channel. We assume tasks never end simultaneously - this assumption is reasonable in a continuous time setting.

Figure 12 shows how a single task with duration $[2; 3]$ is modelled. In this model, we only define the upper

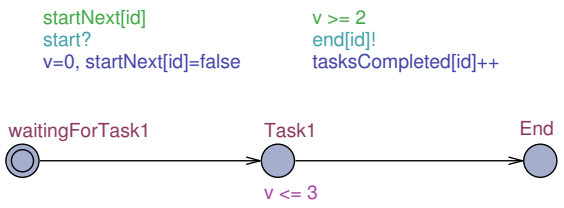


Figure 12: A continuous time task modelled in UPPAAL.

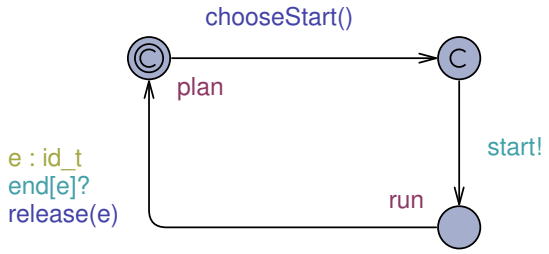


Figure 13: A *fp* scheduler for a continuous time DPA modelled in UPPAAL.

and lower bound of the task. The UPPAAL simulator chooses uniformly amongst the possible values.

Since only one task ends at a time, the **release** function is made so that it requires an argument and releases the resources used by the task completed by the SDPA given. Figure 13 shows the *fp* scheduler that starts these tasks. Starting tasks is performed in the same way as in the discrete model. When a task ends, it sends a signal through a channel only used by one SDPA. The scheduler waits for an *end* signal on all end channels and saves the index of the SDPA sending the signal. This is given as an argument to **release**.

5 Experiments

PRISM can verify properties and return an exact result. UPPAAL can verify the same type of properties using hypothesis testing, but cannot guarantee a correct result. Both tools can approximate the probability of some property using simulations. In this section we test the performance of the UPPAAL and PRISM translations. Before undergoing the tests we consider a small language to describe DPAs in. Then we consider a random generator that outputs DPAs in this language.

5.1 DPA language

A small language called the DPA language has been developed. Consider the following example:

```
1  dpa{
2        resources{res1=10,res2=5}
3        sdpa S1
4        {
5           task{ duration[2,3]
6                 resources[res1=7]}
7           task{ duration[5,7]
8                 resources[res1=6]}
9            }
10       sdpa S2
11       {
12          task{ duration[1,3]
13                resources[res1=4]}
14          task{ duration[4,8]
15                resources[res1=3,res2=4]}
16       }
17    }
```

The above snippet shows an example of a DPA expressed in the DPA language. Every DPA start with dpa {, followed by a resource declaration in line 2 - this represents the resource vector of available resources. Lines 3-9 and 10-16 declare two SDPAs. Lines 5-6 define a single task with duration $[2;3]$ and requires 7 units of resource type res1. Tasks are executed in the order they appear in the SDPA.

5.2 Generating random DPAs

We have developed a small tool to create DPAs in the DPA language. This tool generates random models based on the following parameters:

- Number of SDPAs n,
- number of tasks k,
- minimum duration of tasks min_d,
- maximum duration of tasks max_d,
- minimum usage of each resource type min_r,
- maximum usage of each resource type max_r, and
- number of resource types m.

The DPA generator creates a DPA with n SDPAs with k tasks each. The tasks each have a duration

interval $[a, b]$ where $min_d \leq a \leq b \leq max_d$. The generated DPA has m resources each with a size between $\min_r$ and max_r. The duration (a, b) of a task is found by first extracting a uniformly from the set $\{min_d, \min_d + 1, \ldots, max_d - 1\}$ and afterwards extracting b be uniformly from the set $\{a, a + 1, \ldots, \max_d\}$.

The resource demand of each task is chosen at random. The goal is to create interesting models where multiple tasks (preferably more than 2) are running simultaneously. This means that tasks should seldom require all resources at once, since this prevents all other tasks from running. It should not be disallowed either, since bottleneck situations, where a single task is requiring all resources, can occur in real world problems. In the following we explain how these choices are made.

Choosing size of resource types The size of each resource type is extracted uniformly from the set $\{min_r, min_r + 1, \ldots, \max_r\}$.

Choosing resource demand of tasks A real number between 0 and 1 is chosen with respect to a beta distribution with parameters 4 and 5. The probability density function of this distribution is shown in Figure 14

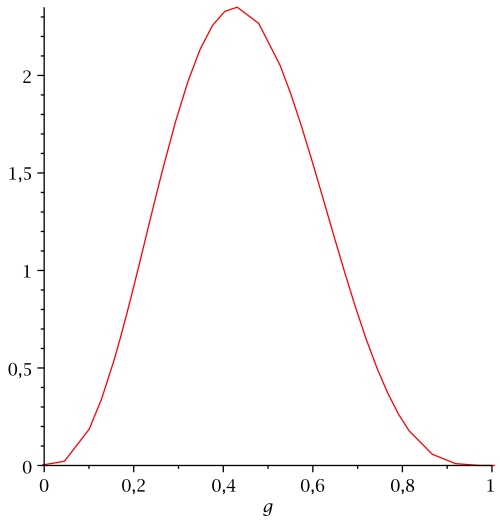


Figure 14: The beta distribution used by the DPA generator for selecting the number of resource types required by a task.

This number is multiplied by m and rounded up to nearest the integer. This finds a number m' between 0 and m. From all the resource types m' are chosen (uniformly) and for each of these, the amount needed by the task is found using a beta distribution with parameters 2 and 4 [6, Definition 6.9.2]. See Figure 15.

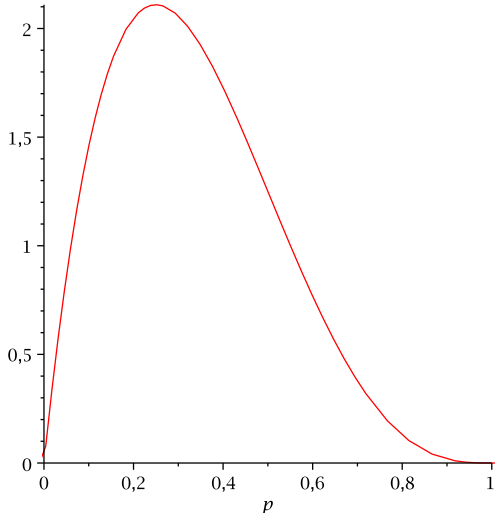


Figure 15: The beta distribution used by the DPA generator for selecting the amount required of a specific resource type.

Two beta distributions are used to find the resource usage of each task. The rationale behind this is as follows: The first beta distributions mean value is 44,4 % of the resource types and the second distribution mean value is 33,3 % of the resources. This allows several tasks to run concurrently, since tasks that require all resources of a specific type are rare. Furthermore, since only 33,3 % of the resources of a type are chosen on average, several tasks can be started before the resource are used up.

5.3 Test setup

When we perform an approximation test, we set the confidence interval to $\delta = 0.05$ and let the confidence be 0.95. When we perform a hypothesis test, we let the probability of type 1 and 2 errors be $\alpha = \beta = 0.05$. The size of the indifference region is set to 0.05. PRISM can not run a hypothesis test using simulation. Instead it can perform verification, always providing a correct answer. We perform verification in PRISM with the sparse engine.

When performing a hypothesis test, a property is needed. The properties we use specify a time limit and a probability of all SDPAs ending before the time limit is reached. In the following, when we perform a hypothesis test or PRISM verification, the query for a DPA is generated as follows: An approximation is run on the continuous time model in UPPAAL, approximating the probability that all SDPAs end within a certain time limit. The time limit chosen is $max_d \cdot n \cdot k$, which is an upper bound on the time it takes for all SDPAs to end. UPPAAL returns a histogram as in Figure 16 containing the actual time (specified in small intervals) spend to end all SDPAs and the frequency of each outcome. We then find the time t at which point 60 % of the runs were completed. We then create the following query: *Is the probability that all SDPAs end within t time units greater than 40 %?* We use the same query for both the discrete and continuous models.
The reason that the query asks for "greater than 40 %" is that the hypothesis testing is not good at testing a hypothesis $\mathcal{P}_{\bowtie\theta}(\gamma)$ when the true probability of γ is close to θ. The confidence interval is set to $\delta = 0.2$ when generating a query. Hereby we reduce the number of simulations UPPAAL performs before terminating. This is acceptable since we only consider the histogram containing simulation data and not the probability returned by UPPAAL.

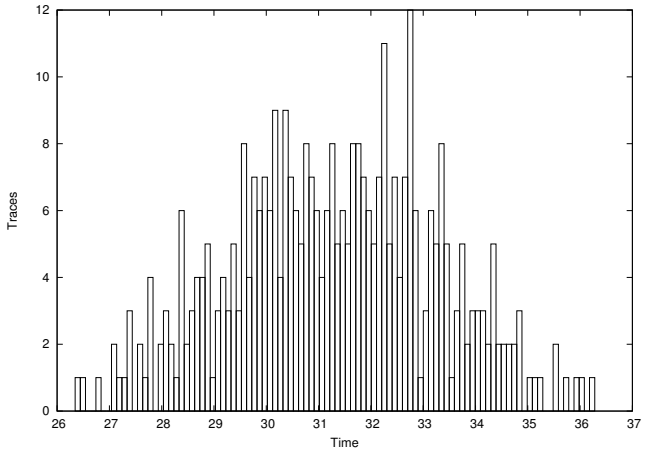


Figure 16: An example of a histogram generated from the outout from UPPAAL.

We perform the following tests:

5.3.1 Duration Test

The goal of this test is to see the effect of increasing the size of the duration interval. We created two SDPAs without resources. The first has $n = 4$, $k = 3$ and the second has $n = 20$, $k = 20$. The duration interval is fixed such that every task has the duration $[2^l, 2 \cdot 2^l]$, where $l \in \mathbb{Z}_{>0}$. We test with different values of l. For each model, we perform a PRISM verification, PRISM approximation, UPPAAL approximation and UPPAAL hypothesis test and measure the time spend.

5.3.2 Performance test - small DPAs

In this test we create random models using the DPA generator with n and k values less than 9 and the following parameters: $min_d = 1$, $max_d = 10$, $min_r = 1$, $max_r = 10$ and $m = 3$. For each model generated, we perform a PRISM verification, PRISM approximation, UPPAAL approximation, UPPAAL hypothesis test and measure the time spend.

5.3.3 Performance test - medium DPAs

Parameters used: $n \leq 40$, $k \leq 40$, $min_d = 1$, $max_d = 40$, $min_r = 1$, $max_r = 40$ and $m = 20$. For each model, we perform UPPAAL hypothesis testing and measure the time spend.

5.3.4 Performance test - large DPAs

Parameters used: $n \leq 55$, $k \leq 55$, $min_d = 1$, $max_d = 80$, $min_r = 1$, $max_r = 80$ and $m = 40$. For each model, we perform UPPAAL hypothesis testing on the continuous time model and measure the time spend.

5.4 Results

In the following, we split the results into approximation results and hypothesis/verification results. We write UPPAAL$_d$ (resp. UPPAAL$_c$) to denote the discrete (resp. continuous) translation. All results are presented in seconds. All results are mean values of 20 runs. For hypothesis testing and verification, the same query is used in all 20 runs.

Duration	PRISM	UPPAAL$_d$	UPPAAL$_c$
$[2, 4]$	4.5	.6	.5
$[4, 8]$	6.8	.7	.5
$[8, 16]$	10.1	.8	.5
$[16, 32]$	15.7	.8	.5
$[32, 64]$	28.1	.9	.5
$[64, 128]$	57.9	1.0	.5

Table 1: Approximation result of the duration test with $n = 4$ and $k = 3$.

Duration	PRISM	UPPAAL$_d$	UPPAAL$_c$
$[2, 4]$	2.6	.1	.1
$[4, 8]$	19.6	.1	.1
$[8, 16]$	274.4	.1	.1
$[16, 32]$	-	.1	.1
$[32, 64]$	-	.1	.1
$[64, 128]$	-	.1	.1

Table 2: Verification and hypothesis test result of the duration test with $n = 4$ and $k = 3$. Entries with "-" indicate the test took longer than 300 seconds and was cancelled.

Duration	UPPAAL$_d$	UPPAAL$_c$
$[8, 16]$	72.4	52.2
$[16, 32]$	95.0	52.4
$[32, 64]$	112.8	53.3
$[64, 128]$	129.4	52.2
$[128, 256]$	146.4	52.1
$[256, 512]$	173.8	52.3

Table 3: Approximation result of the duration test with $n = 20$ and $k = 20$.

Duration	UPPAAL$_d$	UPPAAL$_c$
$[8, 16]$	8.3	2.0
$[16, 32]$	8.0	2.2
$[32, 64]$	5.1	1.8
$[64, 128]$	5.2	1.6
$[128, 256]$	8.1	1.8
$[256, 512]$	11.6	1.8

Table 4: Hypothesis test result of the duration test with $n = 20$ and $k = 20$.

n	k	PRISM	UPPAAL$_d$	UPPAAL$_c$
2	6	5.0	.8	.6
3	6	9.9	1.3	1.0
4	6	17.2	1.9	1.4
5	6	30.6	2.7	2.1
6	6	46.0	3.4	2.6
6	2	9.4	1.1	.9
6	2	10.0	1.0	.8
6	3	15.0	1.7	1.4
6	4	23.6	2.2	1.7
6	5	36.0	2.9	2.3
7	7	79.6	4.9	3.9
8	8	137.8	6.9	5.5
9	9	232.0	9.7	8.1

Table 5: Approximation result of the small model test.

n	k	PRISM	UPPAAL$_d$	UPPAAL$_c$
2	6	5.4	.1	.3
3	6	54.0	.1	.1
4	6	-	.2	.2
5	6	-	.2	.2
6	6	-	.2	.3
6	2	91.3	.1	.1
6	3	-	.1	.1
6	4	-	.2	.2
6	5	-	.2	.2
6	6	-	.4	.2
7	7	-	.2	.3
8	8	-	.4	.3
9	9	-	.4	.4

Table 6: Verification and hypothesis test result of the small model test. Entries with "-" indicate the test took longer than 300 seconds and was cancelled.

n	k	UPPAAL$_d$	UPPAAL$_c$
20	40	23.5	19.1
25	40	39.7	31.9
30	40	42.8	37.1
35	40	98.0	66.0
40	40	98.9	91.0
40	20	47.1	40.8
40	25	52.4	42.9
40	30	70.5	66.1
40	35	108.2	72.2

Table 7: Results of the medium test.

n	k	UPPAAL$_c$
40	55	219.5
45	55	247.8
50	55	323.8
55	55	390.3
55	40	307.0
55	45	294.8
55	50	342.7

Table 8: Results of the large test.

5.5 Discussion

Duration test

The results in Table 1 and 2 show that increasing the duration interval slows both the PRISM approximation and verification. UPPAAL's is too fast on this model to conclude anything about scalability.
Interestingly, Table 3 shows that the UPPAAL approximation is affected in the discrete case, but not in the continuous time case. A possible explanation is that the increased size of the model (due to the number of edges) leads to longer loading times. The largest discrete time model uses 16 MB space, and takes ≈7 seconds to load by UPPAAL. This does not account for the entire difference. Another possible explanation is that UPPAAL becomes slower due to the many edges of the model.
Table 4 shows that the results of the hypothesis test vary a lot, and the only observation we make is that the continuous case seems to be unaffected by the increase in duration intervals.

Performance test - small

The results in Table 6 show that PRISM verification does not scale very well with model size. The results in Table 5 indicates that the PRISM approximation stops being efficient when $n = 9, k = 9$. The result of the hypothesis test of UPPAAL$_c$ for the duration interval $[2, 4]$ relates to the result of Bozga et al. [1]. They show how to verify properties concering untimed behaviour by computing the exact probability of an untimed behaviour. To cover 80 % of the possible behaviours, their approach takes 355.17 seconds compared to 0.1 second using our approach. Note that the two approaches cannot be compared on the same basis - their result is precise and not subject to the randomness of simulations. Nonetheless, this is a strong argument in favour of using statistical modelchecking.

Performance test - medium

The result of the tests are shown in Table 7. It shows that the continuous translation is faster than discrete translation in all cases. Furthermore, it seems that a larger value of n slows the hypothesis test more than a large value of k. For instance, $n = 40, k = 35$ lasts longer than $n = 35, k = 40$. All results in the table support this observation.

Performance test - large

The result of the tests are shown in Table 8 and show that the continuous time translation can be statistically verified effectively and efficiently. The observation from the previous test is also supported by this data.

5.5.1 Possible errors

Some of the results do not seem consistent. We examine the possible causes. One of the main factors is that the model is generated by a random generator.

The generator can assign very small or large duration intervals, and this can influence the discrete translations (as shown in the duration test).
The number of transitions in a UPPAAL or PRISM run can also vary due to the randomness of the model. If several tasks start (or end) simultaneously, this reduces the number of transitions made. This can be influenced by the resource requirements generated by the DPA generator.
Measurements of running time are not precise when the running time is less than a second. For most of the results of the small test, the UPPAAL running times are so small we cannot conclude which translation is fastest.
As mentioned before, hypothesis testing does not perform well when the probability asked for is close to the true probability. When this is the case, more runs are required to produce a result. Hence if the query is generated from a biased sample of all runs, it can influence the performance.
Hypothesis testing is also subject to bias by sheer randomness. It is possible to get 50/200 satisfying runs in a situation where we would expect 20/200, which is

enough to conclude that the hypothesis holds. Hence the number of runs required varies and influences the running time a lot. For instance, consider the discrete translation in Table 7. The case $n = 40, k = 40$ is faster than the case where $n = 40, k = 35$. On average, number of runs performed on the discrete translation was 209.9 for $n = 40, k = 40$, and 261.4 for $n = 40, k = 35$.

6 Conclusion

In this paper we have given a discrete time semantics for Duration Probablilistic Automata with a fixed priority scheduler. We have also given an informal introduction to the semantics in a continuous time setting. Duration Probabilistic Automata have been translated into a PRISM model in a discrete time setting and into UPPAAL Timed Automata in both continuous and discrete time settings. This gave us a framework for testing how statistical model checking performed compared to ordinary model checking. We also compared the efficiency of the translations against each other.
The tests show that the PRISM translation can only handle relatively small models compared to both of the UPPAAL translations. The tests also show that the continuous translation is more efficient than the discrete UPPAAL translation and can handle up to 55 SDPAs with 55 tasks.
The ability of the tools to approximate the probability of satisfying a formula has also been tested. This test show that PRISM can handle larger models when approximating than when verifying.

6.1 Future work

The DPAs we consider are limited to executing tasks once. In the future extending the DPA formalism to circular executions would allow modelling more complex systems. Another possible extension is to let some tasks produce resources and let other consume resources at a given rate. This poses questions such as *"what is the probability that the available resources never drop below x within y time units"*.
Adding the possibility of producing/consuming resources would also allow the modelling of inter-SDPA dependencies i.e. that the execution of a task in one SDPA requires that a task in another SDPA has been executed.

References

[1] Marius Bozga, Jean-Francois Kempf, Kim G. Larsen, Bruce H. Krogh, and Oded Maler. Comparing schedulers in a probabilistic setting. 2010.

[2] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.

[3] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Prism 2.0: A tool for probabilistic model checking. In *QEST*, pages 322–323. IEEE Computer Society, 2004. ISBN 0-7695-2185-1.

[4] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2): 134–152, 1997.

[5] Oded Maler, Kim G. Larsen, and Bruce H. Krogh. On zone-based analysis of duration probabilistic automata. Submitted to INFINITY 2010, 2010.

[6] Peter Olofsson. *Probability, Statistics and Stochastic Processes*. John Wiley and Sons, 2005.

[7] Håkan L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon University, 2005.

A Fixed priority scheduler implementation in UPPAAL

```
void chooseStart()
{
    int i=0, j=0;
    bool enoughResources;
    int reserved[m];
    while(i < n)
    {
        if(waiting[i])
        {
            j = 0;
            enoughResources = true;
            while(j < m)
            {
                if(resources[j] < resUsage[i][tasksCompleted[i]][j]
                    & resUsage[i][tasksCompleted[i]][j]!= 0)
                enoughResources = false;
                j++;
            }
            if(enoughResources)
            {
                takeResources(i);
                startNext[i] = true;
                waiting[i] = false;
            }
        }
        i++;
    }
}
```

B Full PRISM translation

The model translated is the DPA in Section 5.1.

```
dtmc
module scheduler
 doPayback : bool init false;
 counter : [1..3] init 1;
 planning : bool init true;
 timePassed : [0..21] init 0;
 res2 : [0..5] init 5;
 res1 : [0..10] init 10;

 [] ((!doPayback)&planning&counter=3) ->
      1/1:(planning'=false);
 [] (counter=3&doPayback&(!paybackS1)&(!paybackS2)&planning) ->
      1/1:(counter'=1)&(doPayback'=false);
 [end] ((!planning)&(!(taskS1=3&taskS2=3&true))) ->
      1/1:(counter'=1)&(planning'=true)&(doPayback'=true);
 [delay] ((!planning)&timePassed<21) ->
      1/1:(planning'=false)&(timePassed'=timePassed+1);

 [p1payback2] (counter=1&(!(!doPayback))&res1<=4&true) ->
      1/1:(counter'=counter+1)&(res1'=res1+6);
 [] (planning&counter=1&taskS1=2&(!activeS1)&(res1<6|false)&(!doPayback)&true) ->
      1/1:(counter'=counter+1);
 [startp1] ((!doPayback)&planning&counter=1&taskS1=2&(!activeS1)&res1>=6&true&true) ->
      1/1:(counter'=counter+1)&(res1'=res1-6);
 [] (counter=1&(!(!doPayback))&(!paybackS1)&true) ->
      1/1:(counter'=counter+1);
 [p1payback1] (counter=1&(!(!doPayback))&res1<=3&true) ->
      1/1:(counter'=counter+1)&(res1'=res1+7);
 [] (planning&counter=1&taskS1=1&(!activeS1)&(res1<7|false)&(!doPayback)&true) ->
      1/1:(counter'=counter+1);
 [] (planning&counter=1&taskS1=3&(!activeS1)&(!doPayback)&true) ->
      1/1:(counter'=counter+1);
 [] (planning&counter=1&taskS1<3&activeS1&(!doPayback)&true) ->
      1/1:(counter'=counter+1);
 [startp1] ((!doPayback)&planning&counter=1&taskS1=1&(!activeS1)&res1>=7&true&true) ->
      1/1:(counter'=counter+1)&(res1'=res1-7);

 [p2payback2] (counter=2&(!(!doPayback))&res1<=7&res2<=1&true) ->
      1/1:(counter'=counter+1)&(res1'=res1+3)&(res2'=res2+4);
 [] (planning&counter=2&taskS2=2&(!activeS2)&(res1<3|res2<4|false)&(!doPayback)&true) ->
      1/1:(counter'=counter+1);
 [startp2] ((!doPayback)&planning&counter=2&taskS2=2&(!activeS2)&res1>=3&res2>=4&true&true) ->
      1/1:(counter'=counter+1)&(res1'=res1-3)&(res2'=res2-4);
 [] (counter=2&(!(!doPayback))&(!paybackS2)&true) ->
      1/1:(counter'=counter+1);
 [p2payback1] (counter=2&(!(!doPayback))&res1<=6&true) ->
      1/1:(counter'=counter+1)&(res1'=res1+4);
 [] (planning&counter=2&taskS2=1&(!activeS2)&(res1<4|false)&(!doPayback)&true) ->
      1/1:(counter'=counter+1);
 [] (planning&counter=2&taskS2=3&(!activeS2)&(!doPayback)&true) ->
      1/1:(counter'=counter+1);
 [] (planning&counter=2&taskS2<3&activeS2&(!doPayback)&true) ->
      1/1:(counter'=counter+1);
```

```
 [startp2] ((!doPayback)&planning&counter=2&taskS2=1&(!activeS2)&res1>=4&true&true) ->
      1/1:(counter'=counter+1)&(res1'=res1-4);

endmodule

module p1
  taskS1 : [1..3] init 1;
  clockS1 : [0..7] init 7;
  paybackS1 : bool init false;
  activeS1 : bool init false;

  [end] (activeS1&(!clockS1=0)&min(clockS1,clockS2)=0&taskS1<3&(!paybackS1)&true) ->
      1/1:(clockS1'=clockS1);

  [end] (activeS1&clockS1=0&taskS1<3&(!paybackS1)&true) ->
      1/1:(activeS1'=false)&(taskS1'=taskS1+1)&(clockS1'=7)&(paybackS1'=true);
  [end] ((!clockS1=0)&(!activeS1)&(!paybackS1)&true) ->
      1/1:(activeS1'=false);

  [delay] ((!activeS1)&(!paybackS1)&true) -> 1/1:(activeS1'=false);
  [delay] (activeS1&clockS1>0&taskS1<3&(!paybackS1)&true) ->
      1/1:(clockS1'=clockS1-1);


  [startp1] ((!activeS1)&(!paybackS1)&taskS1=2&true) ->
      1/3:(activeS1'=true)&(clockS1'=5)+
      1/3:(activeS1'=true)&(clockS1'=6)+
      1/3:(activeS1'=true)&(clockS1'=7);
  [startp1] ((!activeS1)&(!paybackS1)&taskS1=1&true) ->
      1/2:(activeS1'=true)&(clockS1'=2)+
      1/2:(activeS1'=true)&(clockS1'=3);
  [p1payback2] (paybackS1&taskS1=3) ->
      1/1:(paybackS1'=false);
  [p1payback1] (paybackS1&taskS1=2) ->
      1/1:(paybackS1'=false);

endmodule
module p2
  taskS2 : [1..3] init 1;
  clockS2 : [0..8] init 8;
  paybackS2 : bool init false;
  activeS2 : bool init false;

  [end] (activeS2&(!clockS2=0)&min(clockS1,clockS2)=0&taskS2<3&(!paybackS2)&true) ->
      1/1:(clockS2'=clockS2);
  [end] (activeS2&clockS2=0&taskS2<3&(!paybackS2)&true) ->
      1/1:(activeS2'=false)&(taskS2'=taskS2+1)&(clockS2'=8)&(paybackS2'=true);
  [end] ((!clockS2=0)&(!activeS2)&(!paybackS2)&true) ->
      1/1:(activeS2'=false);

  [delay] ((!activeS2)&(!paybackS2)&true) ->
      1/1:(activeS2'=false);
  [delay] (activeS2&clockS2>0&taskS2<3&(!paybackS2)&true) ->
      1/1:(clockS2'=clockS2-1);

  [p2payback2] (paybackS2&taskS2=3) ->
      1/1:(paybackS2'=false);
```

```
  [p2payback1] (paybackS2&taskS2=2) ->
      1/1:(paybackS2'=false);

  [startp2] ((!activeS2)&(!paybackS2)&taskS2=2&true) ->
      1/5:(activeS2'=true)&(clockS2'=4)+
      1/5:(activeS2'=true)&(clockS2'=5)+
      1/5:(activeS2'=true)&(clockS2'=6)+
      1/5:(activeS2'=true)&(clockS2'=7)+
      1/5:(activeS2'=true)&(clockS2'=8);

  [startp2] ((!activeS2)&(!paybackS2)&taskS2=1&true) ->
      1/3:(activeS2'=true)&(clockS2'=1)+
      1/3:(activeS2'=true)&(clockS2'=2)+
      1/3:(activeS2'=true)&(clockS2'=3);
endmodule
```

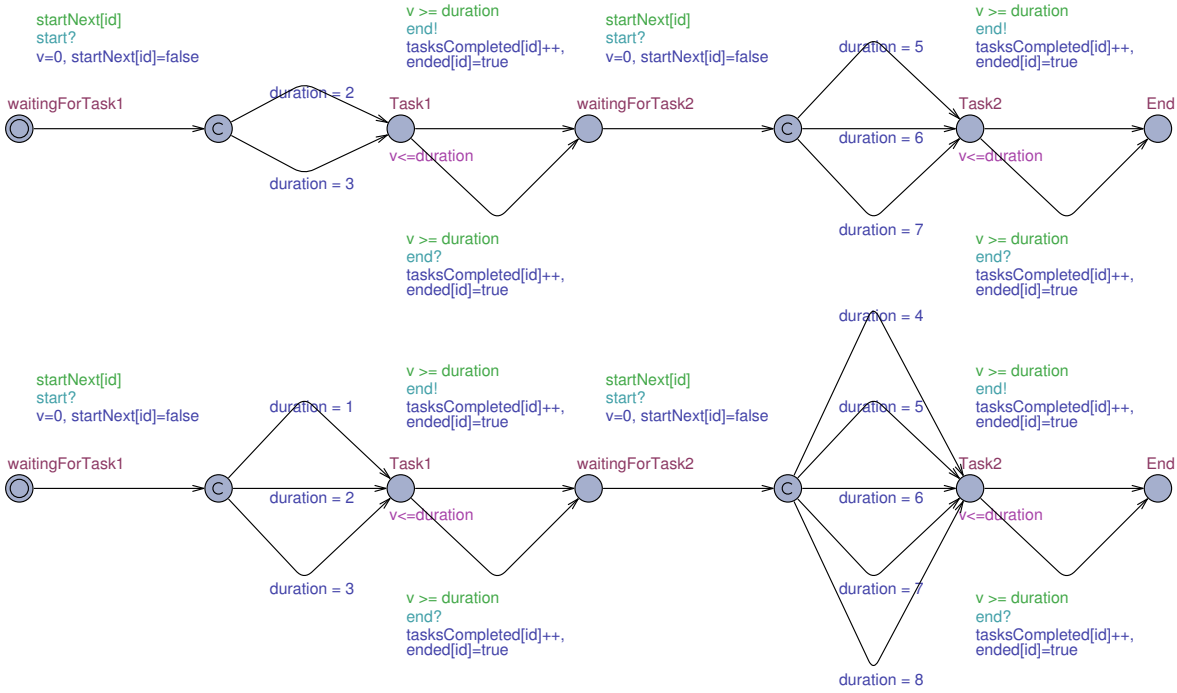
C Full UPPAAL discrete translation

The model translated is the DPA in Section 5.1. The system declarations contains:

```
const int n = 2;
const int k = 2;
const int m = 2;
typedef int[0,n-1] id_t;
bool startNext[n];
int tasksCompleted[n];
broadcast chan start;
broadcast chan end;
bool ended[n];
bool waiting[n]={true,true};

int resources[m]={10, 5};
int resUsage[n][k][m]= {{{7,0},{6,0}},{{4,0},{3,4}}};

void takeResources(id_t sdpa)
{
    int i = 0;
    while (i < m)
    {
        resources[i] -= resUsage[sdpa][tasksCompleted[sdpa]][i];
        i++;
    }
}

void releaseResources(id_t sdpa)
{
    int i=0;
    while(i < m)
    {
        resources[i] += resUsage[sdpa][tasksCompleted[sdpa]-1][i];
        i++;
    }
}

void release()
{
    int i=0;
    while(i < n)
    {
        if(ended[i])
        {
            releaseResources(i);
            ended[i]=false;
            if(tasksCompleted[i] < k)
                waiting[i]=true;
        }
        i++;
    }
}

void chooseStart()
{
    int i=0, j=0;
```

```
    bool enoughResources;
    int reserved[m];
    while(i < n)
    {
        if(waiting[i])
        {
            j = 0;
            enoughResources = true;
            while(j < m)
            {
                if(resources[j] < resUsage[i][tasksCompleted[i]][j]
                                    & resUsage[i][tasksCompleted[i]][j]!= 0)
                {
                    enoughResources = false;
                }
                j++;
            }
            if(enoughResources)
            {
                takeResources(i);
                startNext[i] = true;
                waiting[i] = false;
            }
        }
        i++;
    }
}
```

The generated model consists of a scheduler module (Section 4.2.2) and the following Timed Automata, representing SDPA S1 and S2 in discrete time:



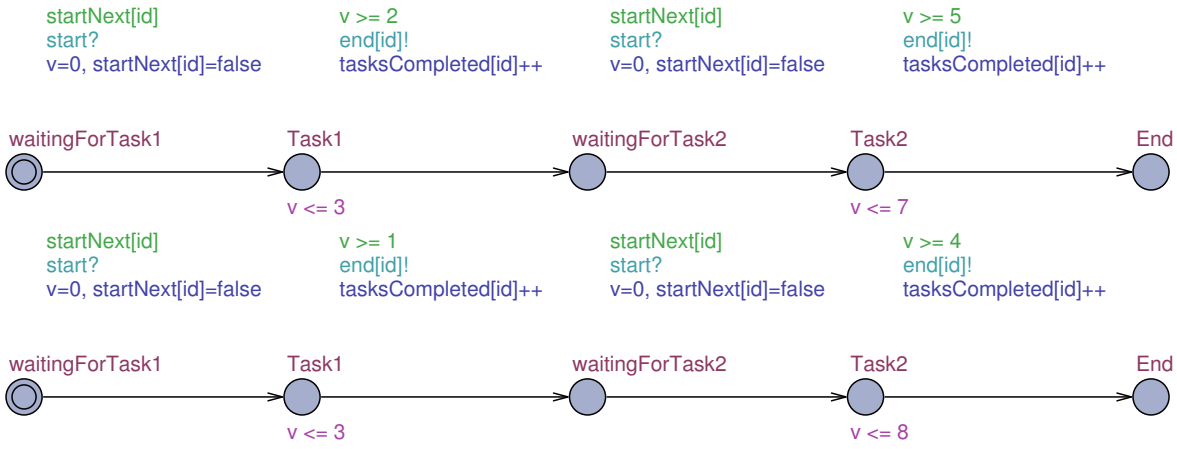
D Full UPPAAL continuous translation

The model translated is the DPA in Section 5.1. The system declarations contains:

```
const int n = 2;
const int k = 2;
const int m = 2;
typedef int[0,n-1] id_t;
bool startNext[n];
int tasksCompleted[n];
broadcast chan start;
broadcast chan end[n];
bool waiting[n]={true,true};

int resources[m]={10, 5};
int resUsage[n][k][m]= {{{7,0},{6,0}},{{4,0},{3,4}}};

void takeResources(id_t sdpa)
{
    int i = 0;
    while (i < m)
    {
        resources[i] -= resUsage[sdpa][tasksCompleted[sdpa]][i];
        i++;
    }
}

void releaseResources(id_t sdpa)
{
    int i=0;
    while(i < m)
    {
        resources[i] += resUsage[sdpa][tasksCompleted[sdpa]-1][i];
        i++;
    }
}

void release(id_t sdpa)
{
    releaseResources(sdpa);
    if(tasksCompleted[sdpa] < k)
        waiting[sdpa]=true;
}

void chooseStart()
{
    int i=0, j=0;
    bool enoughResources;
    int reserved[m];
    while(i < n)
    {
        if(waiting[i])
        {
            j = 0;
            enoughResources = true;
            while(j < m)
            {
```

```
                if(resources[j] < resUsage[i][tasksCompleted[i]][j]
                                    & resUsage[i][tasksCompleted[i]][j]!= 0)
                {
                    enoughResources = false;
                }
                j++;
            }
            if(enoughResources)
            {
                takeResources(i);
                startNext[i] = true;
                waiting[i] = false;
            }
        }
        i++;
    }
}
```

The generated model consists of a scheduler module (Section 4.3) and the following Timed Automata, representing SDPA S1 and S2 in continuous time:



E DPA modelling language grammar

The grammar of the DPA modelling language. The final 3 rules are recognized by a lexical analyser and are specified in EBNF, while the other rules are specified in BNF. Bold text indicates terminal tokens.

Dpa	:=	**dpa {** ResourceDeclaration SdpaList **}**
Sdpa	:=	**sdpa** Name **{** TaskList **}**
Task	:=	**task {** Duration ResourceUsage **}**
		| **task {** Duration **}**
ResourceDeclaration	:=	**resources {** ResourceQuantityList **}**
ResourceQuantityList	:=	ResourceQuantity
		| ResourceQuantity **,** ResourceQuantityList
ResourceUsage	:=	**resources [** ResourceQuantityList **]**
ResourceQuantity	:=	Name = Number
TaskList	:=	Task
		| Task TaskList
SdpaList	:=	Sdpa
		| Sdpa SdpaList
Duration	:=	**duration [** Number **,** Number **]**
Name	:=	Letters Number?
Letters	:=	$[\mathbf{a}-\mathbf{z}, \mathbf{A}-\mathbf{Z}]^+$
Number	:=	$[\mathbf{0}-\mathbf{9}]^+$