Learning Movement Synchronization in Multi-component Robotic Systems

Mohammad Thabet\textsuperscript{1}, Alberto Montebelli\textsuperscript{2}, and Ville Kyrki\textsuperscript{3}

Abstract—Imitation learning of tasks in multi-component robotic systems requires capturing concurrency and synchronization requirements in addition to task structure. Learning time-critical tasks depends furthermore on the ability to model temporal elements in demonstrations. This paper proposes a modeling framework based on Petri nets capable of modeling these aspects in a programming by demonstration context. In the proposed approach, models of tasks are constructed from segmented demonstrations as task Petri nets, which can be executed as discrete controllers for reproduction. We present algorithms that automatically construct models from demonstrations, showing how elements of time-critical tasks can be mapped into task Petri net elements. The approach is validated by an experiment in which a robot plays a musical passage on a keyboard.

I. INTRODUCTION

Programming by demonstration (PbD), also called imitation learning or learning from demonstration, is a recent approach that promises a rapid and efficient way to impart skills to robots [1]. In PbD, robots are taught new skills or tasks simply by having a human teacher (or possibly another robotic agent) perform demonstrations, which they capture using various sensors. The robot then utilizes learning algorithms to learn the task and reproduce it with a measure of adaptability.

Robotic systems can often be considered as composed of multiple parts that operate independently, and some classes of tasks can be considered as a combination of such movements. This is an issue that is left largely unexplored in the literature, as robots are generally treated as atomic entities in the context of PbD. However, by considering each part separately, it becomes possible to learn complex movements of the entire robot as a collection of simple movements of its components, or reuse pre-learned movements of individual parts in new tasks, which can reduce the complexity of the learning process.

One class of tasks that is of special interest is time-critical tasks. These are tasks that require accurate timing of their component actions to be considered successful, analogous to hard real-time processes in computer software. Therefore, a learner must not only learn what to do, but also exactly when to do it and for how long. Furthermore, in multi-component systems, concurrency and synchronization of individual components also need to be preserved.

This paper proposes task Petri nets (TPNs), a modeling framework for multi-component robotic systems. We introduce TPNs as task models capable of capturing the task structure and temporal information as well as synchronization and concurrency requirements between components. In our approach, the trajectory of each component in a demonstration is individually segmented into primitives, and a TPN is constructed from the primitive streams (Fig. 1). The resulting TPN can be used as a PN controller for reproduction, enforcing synchronization between components in the presence of execution delays and allowing for the reproduction of time-critical tasks. We present algorithms that automatically construct a TPN model of a task from multiple sets of time-stamped observations. In a PbD context these observations can arise from segmenting demonstration data from multiple robotic parts into movement primitives. The approach is validated through an experiment in which a robot learns and reproduces a musical passage on a keyboard.

II. RELATED WORK

The prevalent technique to model complex tasks in PbD is to use a two-level hierarchy where a top-level discrete symbolic layer controls the switching behavior of underlying action/movement primitives [2], [3], [4], [5]. In these

\textsuperscript{1}Mohammad Thabet performed the work leading to this paper with the Department of Electrical Engineering and Automation, Aalto University, Finland. He is currently with the Department of Intelligent Hydraulics and Automation, Tampere University of Technology, Finland, mohammad.thabet@tut.fi

\textsuperscript{2}Alberto Montebelli is with the School of Informatics, University of Skövde, Sweden, alberto.montebelli@his.se

\textsuperscript{3}Ville Kyrki is with the Department of Electrical Engineering and Automation, Aalto University, Finland, ville.kyrki@aalto.fi

Fig. 1. Overview of proposed approach.
approaches, the symbolic level is typically modeled using directed graphs or finite state machines. For example, in [5] kinesthetic demonstrations are used to teach a robotic hand and arm to unscrew a light bulb. There, learning the skill consists of constructing an FSM where the states are primitives, and transitions signify switching between primitives, of which only one can be active at a time. Each state is associated with a classifier which has been trained using labeled demonstration data, and decides whether to continue executing the current primitive or switch to the next one based on features obtained from raw sensor data.

PNs have been widely used in robotics, for example in single-robot task specification to coordinate task primitives [6], [7], coordination of dual-arm manipulation tasks [8], and in multi-robot plans [9]. However, PNs have rarely been applied in a PhD setting. In [10], PN models of object placement tasks were learned from demonstration, in which places represent object states and transitions represent motions. At the beginning of a demonstration, the state of each object is represented by a place in the PN. When a motion finishes, it is added as a transition in the net if it is new and has not previously been performed. Object states before and after a motion are added as input and output places to the transition respectively. For imitation, images of the initial and goal states are obtained and transformed into markings of the PN. A reachability graph is then generated and traversed to find the shortest sequence of transition firings from the initial marking to the goal marking.

The issues of modeling time or performing multiple actions simultaneously and synchronizing them have not been addressed in any of the previous approaches. Conversely, our work focuses on modeling tasks with multiple overlapping primitives and strict timing requirements. The main contribution of our work is developing algorithms to construct a TPN model from segmented demonstration data that effectively captures the structure of these tasks. A similar idea has been studied in process mining, where a PN model of a process can be discovered from event logs, in what is known as process discovery [11], [12]. The resulting models are a subset of PNs called workflow nets. However, our approach differs in two major ways. First, process mining approaches do not handle temporal data. Second, workflow nets model actions as transitions, while places represent conditions such as the completion of an action. This is contrary to the design philosophy of our approach, which is derived from control interpreted PNs [13].

III. PROPOSED APPROACH

Modeling time-critical tasks at the symbolic level requires not only capturing the task structure, but also the temporal relationship between task components. Furthermore, for multi-component robotic systems whose components can operate independently, it is imperative to synchronize these components to perform the task correctly and in a timely manner. Similar to other work, we approach the modeling problem as a discrete-event system. However, in order to handle such a requirement, it is necessary to go beyond the practice of sequencing primitives as state machines prevalent in the literature and employ a more powerful modeling framework. We propose Petri nets as the basis of such a framework.

A. Overview of Petri Nets

PNs are a mathematical modeling tool (language) mainly used to model discrete-event dynamic systems. They combine a rigorous mathematical formulation with an intuitive graphical representation, which allows accurate modeling and analysis as well as easy visualization and interpretation of models. Furthermore, they are vastly extensible beyond their basic form, and have been well-studied and analyzed for decades [13].

A Petri net is a five-tuple \( PN = (P,T,W^+,W^-,M_0) \) such that:

- \( P = \{P_0, P_1, \ldots, P_n\} \) is a finite set of places;
- \( T = \{T_0, T_1, \ldots, T_n\} \) is a finite set of transitions;
- \( P \cap T = \emptyset \) i.e., the sets \( P \) and \( T \) are disjoint;
- \( W^- : P \times T \rightarrow \mathbb{N} \) is the input incidence matrix of size \( m \times n \);
- \( W^+ : P \times T \rightarrow \mathbb{N} \) is the output incidence matrix of size \( m \times n \);
- \( M_0 : P \rightarrow \mathbb{N} \) is the initial marking vector of length \( m \).

Places in a PN can be marked with tokens. The collective distribution of tokens in the net is called the marking and represents the state of the system. Places and transitions are connected by weighted arcs whose weights are given in the incidence matrices. Transitions whose input places have sufficient tokens (defined by the input incidence matrix) are enabled and can fire, changing the marking of the net.

Numerous extensions have been introduced to PNs to increase their modeling power. Those relevant to our approach are synchronized PNs, T-timed PNs, and colored PNs [13]. In synchronized PNs, transitions are synchronized on external events and can only fire when these events occur. In T-timed PNs, transitions are associated with time delays and can fire only when a certain time has elapsed since their enabling. Finally, in colored PNs values are attached to tokens, and arcs are associated with expressions that modify these values as tokens move around the net [14].

B. Task Petri Nets

We define a Task Petri Net (TPN) as a timed, synchronized, and colored PN model of a task. The task to be modeled is considered as composed of streams or strings of symbols that denote actions or action/movement primitives which include the wait symbol as shown in Fig. 2(a). TPNs are similar to control interpreted PNs, except that places are associated with symbols, transitions can be timed as well as synchronized, and tokens are colored.

The structure of a TPN consists of a number of main branches equal to the number of system components used in the task as shown in Fig 2(b). The term ‘branch’ here is used to denote a series of mainly single-input, single-output places and transitions in succession. The relationship
between branches and components is one-to-one. There is always one and only one start place in a TPN. This place is initialized with a single token in the initial marking, and spawns at least one branch. All branches can arise from either the start place directly, or another branch, depending on the time the corresponding component starts its first movement relative to other components.

The elements of a TPN are related to those of a task as follows:

**Places:** Places in a TPN are associated with (i) symbols, (ii) repetition counts, or (iii) conditions. Most places in a TPN are associated with symbols such that a token in such a place signifies the execution of the action associated with the symbol. Count places are used as input places for looping transitions, and are initialized with a number of tokens equal to the number of repetitions in the loop. Finally, some places are associated with a condition such as the execution ready condition as in the start place.

**Tokens:** Tokens in symbol places are colored, where the color of a token carries information that can modulate the execution of the action associated with its containing place. An example of such information is the nominal duration of the action. Tokens in condition and count places are uncolored.

**Transitions:** Transitions in a TPN are either synchronized on external events, or timed. In the first case the synchronization event dictates when an action should begin or end, while in the second case the time delay dictates how long the system should wait before an action is performed.

There are three main aspects of tasks that a TPN is required to model. The first of these is the task structure, which refers to the composition of a task in terms of its component movement primitives (or symbols) and their order. The second aspect is concurrency requirements and synchronization, and relates to which primitives of different components need to execute simultaneously. And thirdly, the temporal aspect which refers to the time in which primitives are required to be executed and their duration.

To model the basic task structure, each symbol stream is mapped to one branch of the TPN, reflecting the concurrent execution of both streams. The order of primitives is preserved in the sequence of places in each branch, reflecting the sequential execution of primitives in each symbol stream. Furthermore, repeating primitives in the task are modeled as loops in the TPN. For example, the fact that the sequence \((a, c, \emptyset)\) is repeated in the upper symbol stream in Fig. 2(a), is modeled by the looping transition \(T7\) in the TPN of Fig. 2(b). The number of repetitions is given by the number of tokens in the accompanying place \(P7\). During execution, looping transitions can be in conflict with other regular transitions, in which case looping transitions are always favored.

For tasks involving multiple robotic components moving simultaneously, it is not sufficient to simply sequence the primitives. It is assumed that during a wait symbol, the component is waiting for other components to either start...
executing a certain primitive or finish executing one. For example, in Fig. 2(a) there is a concurrency requirement that the last symbol \( b \) in the second stream be executed while \( C \) is executing in the first stream during reproduction. If the execution of any primitive preceding \( b \) in the same stream is delayed significantly, \( b \) might be executed after \( C \) has finished, thus failing the concurrency requirement. This requirement is modeled in the TPN of Fig. 2(b) as an arc from \( T_6 \) to \( P_5 \).

The time at which a symbol is executed is decided by its preceding symbol. If a symbol is preceded by a non-wait symbol, then it executes immediately after the preceding symbol finishes. In this case, the input transitions of the place associated with that symbol are synchronized on the completion event of the execution of the preceding symbol. On the other hand if it is preceded by a wait symbol, then it waits for a certain amount of time. In a TPN, timed transitions are used to introduce delays between primitives. Notice that the branch corresponding to any stream that starts later than other streams begins with a place associated with a wait symbol, which is also followed by a timed transition.

The duration of execution of non-wait symbols is decided by the color (i.e., value) of the token inside its associated place. The arc expressions in the TPN reflect the relationship between the durations of any two successive symbols in a branch. For example if the duration of a symbol is half that of its predecessor, then the expression on the output arc of the transition between them is 0.5. This means that if a token in the preceding place that has a value of 1 is consumed, then the deposited token in the succeeding place will have a value of 0.5. Consequently, the temporal scale of the entire reproduction is decided by the value of the token in the start place in the initial marking.

IV. LEARNING TASK MODELS

Before the construction of the TPN, all trajectories in the demonstration are individually segmented into discrete symbols that represent segments in these trajectories. A symbol is primarily a representation of a primitive-goal pair, represented by the id of the component that originated it, the id of the movement primitive that represents the segment, and the goal state of the primitive (or endpoint of the segment). It also contains the start and end times of the movement, corresponding to the time-stamps of the first and last data points in the segment respectively. Thus, it also implicitly contains information on the duration of the primitive.

After segmentation, all the symbol streams are collected into a stream bundle object. This object provides the necessary logical encapsulation to process the different streams simultaneously. Most importantly, it provides the method \( \text{getNextSymbol()} \) that returns symbols in temporal order regardless of which component stream they belong to.

The process of building the TPN and later executing it is general and independent of the primitive framework used, as long as the resulting symbols implement a particular interface. Building the TPN consists of three stages: naïve PN construction, concurrency enforcement, and folding.

A. Naïve Petri Net Construction

A naïve PN representation of the task is first constructed to serve as a basis for further refinement. This PN consists of \( n \) branches corresponding to \( n \) symbol streams, each consisting of a series of places representing symbols in a stream. The algorithm, shown in Alg. 1, starts by creating a start place and an output transition for it, which will be called the start transition. Afterwards, a place is created for the first symbol in the stream bundle as an output place to the start transition. For each subsequent symbol in the bundle for the same component, a new place is created, as well as a transition from the previous place to the new symbol, creating the component branch in the process. When the first symbol in a stream is encountered later on, a leading wait place is created as an output place to the input transition of the last place created, thus creating a new branch. Another place is also created for the symbol in question, and a transition is created from the leading wait place to it. Subsequent symbols are treated normally as in the first branch. This process continues until all symbols in the bundle have been processed.

![Algorithm 1: Constructing a naïve Petri net.](image)

Whenever a place or transition is created, it is linked with the previous element in the same branch with an arc, which always has a weight of 1. Furthermore, each output arc of a transition has an expression equal to the duration of the symbol associated with the output place divided by that of the input place. Note that at this stage, all transitions have a single input place, whereas transitions that spawn branches may have more than one output place.

While constructing the PN, lists of all created places, transitions and arcs are stored. All elements in a list have ids that are used to consistently reference them in other elements. Places contain the ids of their input and output transitions, while transitions contain the ids of their input and output places. Similarly, arcs contain the ids of their originating and terminating nodes. Throughout the TPN construction process, the ids of places always have the same order as
the temporal order of their associated symbols in the stream bundle. This facilitates temporal comparisons required in certain operations as will be evident later.

**B. Concurrency Enforcement**

After a naïve PN is obtained, the next stage is to enforce concurrency between its branches. In this stage, an algorithm (Alg. 2) uses wait places (i.e., places associated with wait symbols) to either set transition delay times, or force the execution of succeeding symbols to wait for other symbols in other branches to start executing. The algorithm goes through all the places of the naïve PN to find wait places. For each wait place, there are two scenarios possible. If the temporally succeeding place is in the same branch, the output transition of the wait place is in a different branch, an arc is created from the input transition of that temporally preceding place to the wait place. Furthermore, the output transition of the wait place is set as a timed transition and its delay is set to the duration of the wait symbol. This means that the component just has to wait for the duration of the wait symbol. On the other hand, if the temporally succeeding place is in a different branch, an arc is created from the input transition of that temporally preceding place to the wait place. Furthermore, the output transition of the wait place is set as a timed transition, and its duration is given by the time difference between the start of the symbol to be waited for and the start of the symbol that is waiting. This relates to the case that the place following the wait place in the same branch should wait for the temporally preceding place to become active.

```
Require: naïve PN N
P = getPlaceList(N)
for i = P.start to P.end do
  if P(i) is a wait place then
    if P(i) and P(i + 1) are in the same branch then
      t.delay = getSymbolDuration(P(i))
    else
      n = index of the place after P(i)
      in the same branch
      t = getSymbolDuration(P(n))
      add arc from t to P(n)
      t = getOutputTransition(P(i))
      add arc from t to P(n - 1)
      t.delay = getSymbolDuration(P(n - 1))
      getSymbolStartTime(P(n))
    end if
  end if
end for
```

Algorithm 2: Enforcing concurrency in a Petri net.

**C. Folding**

Folding is the third and final stage in the construction of a TPN, in which repeating sequences in the net are replaced with loops. Since the temporal order of symbols is reflected in the order of places, repeating sequences of places correspond directly to repetitions in symbols. A sequence is considered repeating if it appears again immediately after it ends with nothing in between. Furthermore, only repetitions across all branches are considered true repetitions and are consequently processed.

The algorithm used in this stage is similar in structure to that used in [5]. The algorithm starts by searching the entire TPN for repetitions. Once a repetition is found, all its constituent places are merged with their counterparts in later instances of the repetition, and all lists and structures are updated to reflect the changes. The process then starts again, and keeps running until no repetitions are found. For each repetition found, a new count-typed place is created as an input place to the transition between the first and last places of the repeating sequence. In the initial marking, this count place is initialized with a number of tokens equal to the number of instances of the repetition found.

When searching for repetitions, the algorithm (Alg. 3) operates on the list of places. It starts by looking for repeating sequences of length equal to half that of the entire list. If none are found, the length is decremented by one, and this process is repeated until the length is less than two places. This effectively favors longer repetitions over shorter ones. For a sequence of length l, the algorithm searches for repetition by comparing each place P_k in the list with place P_{k+l}. If the two places are equal, then P_{k+l} is compared with P_{k+l+1} and so forth until P_{k+l-1} is compared with P_{k+2l-1}. Any two places are considered equal if their associated symbols are equal, and any two symbols are considered equal if they have the same component and primitive ids and both their goal states and durations are roughly equal within a certain tolerance. When a pair of places being compared are not equal, the starting i is incremented by one, and the process starts again. The process stops when (i + 2l) > l where L is the length of the list, at which point the sequence length l is decremented and the process is restarted. Whenever two adjacent sequences match, the starting i in the search window is incremented by l to search for other instances of the sequence.

```
Require: PN N
P = getPlaceList(N)
repetitionCount = 0
for l = P.length/2 to 2 do
  for i = P.start to P.end - 2l + 1 do
    k = i
    while [P(k) to P(k + l - 1)] = [P(k+l) to P(k+2l-1)] do
      repetitionCount +=
      k = k + l
    end while
  end for
end for
```

Algorithm 3: Finding repetitions in a Petri Net

When merging two places, the one with the higher id is absorbed into the lower-id one and gets marked for deletion, since a lower id corresponds to an earlier symbol. Each input or output transition of the absorbed place is compared to those of the absorbing place; if an equivalent is found then
it gets marked for deletion, otherwise it becomes an input or output transition of the absorbing place by having the ids of the input and output nodes changed accordingly in both the place and the transition. Two transitions are considered equivalent if all their input and output places are equivalent. All arcs in the arcs list get updated to reflect these changes. Finally, the deletion of all elements that were marked for deletion takes place when the lists get updated after each folding process.

V. EXPERIMENTS

In this section we present an experiment to validate our approach. The task chosen for the experiment was playing a small musical passage on a keyboard. The purpose of the experiment was to evaluate the capability of the system to preserve temporal information and structure in tasks, and to enforce synchronization and concurrency requirements in several robotic components in the presence of delays in execution. Keyboard playing is a prime example of a task that places strict requirements on these aspects. The arm and the fingers have to be synchronized almost perfectly in order to hit the correct note (and only the correct note) in a very limited time window. Moreover, the duration of notes and silence in between have to be preserved for the reproduction to be considered successful.

In addition, keyboard playing offers a convenient way of systematically evaluating the performance of the task reproduction. A MIDI-enabled musical keyboard can send MIDI signals of note on/off events, which can be recorded as MIDI tracks by a computer and analyzed. The tracks contain the sequence number and time-stamps of the events recorded, among other information. Thus, by comparing different MIDI tracks, it is possible to compare the performance of the reproduction with that of the original demonstration with a temporal resolution in the order of milliseconds. This has the advantage of bypassing a non-trivial problem that would be present in most other task domains: extracting events on such a temporal scale would be significantly more difficult, assuming they can even be easily defined in the first place.

A. Experimental Setup

The setup used for the experiment consisted of a 4-DoF BarrettHand BH8-282 robotic hand attached to a 7-DoF KUKA LWR4+ lightweight robotic arm. For segmentation and learning, the system was divided into 4 components: the arm and each of the 3 fingers of the hand. The arm trajectory was recorded and segmented in 6-dimensional task space, while those of the fingers were in one-dimensional joint space. To allow for kinesthetic demonstrations, gravity compensation mode was activated for the arm, while a specialized compliance controller was developed and used for the hand.

Since the focus of our work is on learning task structure as well as synchronization and timing requirements, we opted to use a simplistic primitive and segmentation framework based on simple linear movement primitives. This also serves to demonstrate that some complex tasks can be learned as a combination of simple movements of different components. Trajectories were segmented into ‘moving’ and ‘stopped’ segments by cutting the trajectories at the zero-crossing points of the velocity (with some hysteresis), and symbols were generated from the segments. The resulting symbol streams were filtered to obtain the final streams by removing symbols generated from erroneous segmentation points. During reproduction, execution symbols for the arm obtained from the PN controller were converted into joint-space trajectories with a trapezoidal velocity profile using inverse kinematics. Execution symbols for the fingers were converted into velocity commands for the finger joints.

B. Metrics

For quantitative evaluation of the performance of the reproduction, two metrics were used. Firstly, event time error (ETE) measures absolute delays in events. For some reproduction of a demonstration containing \( n \) events, the ETE of the \( i \)-th event is defined as:

\[
ETE_i = |t_i - t_0| - (t'_i - t'_0)|; \quad i = 0, 1, 2, \ldots, n-1,
\]

where \( i \) is the event index starting from 0 for the zeroth event (the very first event), \( t_i \) is the time of the \( i \)-th event in the reproduction, and \( t'_i \) is the time of the \( i \)-th event in the original demonstration. Naturally, the ETE is zero for the zeroth event, since the latter only serves to synchronize the clocks of a demonstration and its reproduction.

The second and most important metric is the inter-event time error (IETE), and relates to jitter. The IETE of the \( i \)-th event is given by:

\[
IETE_i = |ETE_i - ETE_{i-1}|; \quad i = 1, 2, \ldots, n - 1.
\]

IETE describes how well the temporal relationship between events is preserved in a demonstration, and is required to be as low as possible for all events. Since it is given by the difference in ETE of two successive events, this entails that if the ETE of an event increases, the ETE of all subsequent events must increase by the same amount.

C. Experimental Procedure

A MIDI keyboard was placed underneath the hand, and connected to a computer running a music sequencer that records MIDI tracks. The musical passage played on the keyboard consisted of fourteen notes across five keys. The demonstration was made kinesthetically as illustrated in Fig. 3(a). While the demonstration was being performed, the music sequencer recorded the MIDI track of the demonstration. The trajectories were then segmented and the TPN constructed, initialized and executed. The TPN was initialized and executed three times at different speeds. The token in the start place was first initialized with a value of 1 for execution at the original demonstration speed, then a value of 0.5 for execution at double speed, and finally a value of 0.33 for execution at triple speed. MIDI tracks of the reproduction at each execution speed were recorded. Furthermore, the delay in the execution of all primitives in each reproduction was calculated and recorded for further analysis. This execution
delay is the difference between the nominal duration of execution of a primitive as given in the TPN, and the actual duration as executed by the component. In all reproductions, the TPN was updated at a rate of 300 Hz.

In addition to reproductions by executing the TPN, reproductions by simple replay of the demonstrated trajectory were also recorded at original, double, and triple speeds. An impedance controller with relatively low stiffness was used for the arm in trajectory replay for safety reasons, also introducing minor delays in reproduction. This allows comparison of both methods of reproduction in the presence of delay, as there is always some delay in the execution of primitives in the TPN reproduction. This delay in primitive execution is due to approximations made in the primitive executors and the delay in the signal path of the non-real-time part of the system, as well as the inherent inaccuracy in physical robotic systems.

D. Results and Analysis

For analysis of the results, all recorded MIDI tracks were parsed to extract event times. Since the musical passage played contained fourteen notes, each MIDI track contained twenty-eight events. The ETE and IETE were calculated for all events. Data for the original demonstration at double and triple speeds used to calculate the metrics were obtained by multiplying the event times of the original demonstration by 0.5 and 0.33 respectively.

Figure 4 shows the result of the reproduction at original, double, and triple speeds, while Table I gives means and standard deviations of the IETE (\( \mu_{\text{IETE}} \) and \( \sigma_{\text{IETE}} \)), as well as those of the arm and hand execution delays (\( \mu_{\text{ED}} \) and \( \sigma_{\text{ED}} \)). The reproduction at the original speed by the TPN generally compensated for delays in execution and kept the IETE low as shown in Fig. 4(a). However, owing to the trapezoidal velocity profile of the arm trajectories, at such a low speed the fingers did not hit the keys with sufficient force as the arm slowed down as it approached its goal position. For the same reason, the arm did not release the keys fast enough as it ramped up to the maximum speed from rest. This resulted in further delays in key on/off events as exemplified by peaks in the ETE of the TPN reproduction at events 9, 19 and 25 when the pressing and releasing force was fully supplied by the arm. These delays could not be inferred from the overall execution delay of a primitive, and thus could not be fully corrected by the TPN. Furthermore, the average arm execution delay was positive and large, while that of the hand was negative with large \( \sigma_{\text{IETE}} \), which led to complex patterns of interaction that sometimes resulted in negative overall delays. Nevertheless, the IETE of the TPN reproduction was still comparable to that of the trajectory replay.

The effects of the velocity profile were alleviated to some extent in the reproduction at double and triple speeds as shown in Fig. 4(b) and Fig. 4(c) respectively, although minor spikes can still be seen at events 9, 19 and 25. Here, the TPN almost always compensated for delays. As a result, and since the average execution delay of both the arm and the hand was positive, ETEs were increasingly rising. This is required to keep the IETE low as a result of Eq. 2. It can be seen that the IETE for the TPN reproduction always plummets after it peaks, as the TPN senses delay and adjusts succeeding primitives accordingly. This is in contrast to the IETE of the trajectory replay, where double peaks are not uncommon. Although the average ETE of the TPN is significantly higher

![Fig. 3. Demonstration and reproduction of the keyboard-playing task. (a) the task being demonstrated kinesthetically, and (b) the robot reproducing the task.](image1)

![Fig. 4. Results of the reproduction with trajectory replay at low stiffness. (a) at normal speed, (b) at double speed, and (c) at triple speed.](image2)
than that of the trajectory replay, the average IETE of the TPN was almost equal to that of the replay at double speed and 26% less at triple speed. Furthermore, although $\mu_{ED}$ and $\sigma_{ED}$ are significantly larger for the TPN reproduction at triple speed than at double speed, the TPN performance was superior at triple speed. This is reflected in that the average ETE was less at double speed than at triple speed, while the average IETE was greater. The behavior can be attributed to the fact that the effects of the velocity profile are less pronounced at higher speeds.

VI. CONCLUSION

In this paper we proposed a modeling framework for robotic tasks based on PNs. We introduced the concept of TPNs as task models in PbD capable of modeling time-critical tasks for multi-component robotic systems, and presented algorithms to automatically construct them from segmented and time-stamped demonstrations. We showed how TPNs can capture the synchronization requirements between multiple robotic components in tasks by exploiting wait symbols. The resulting TPN can be executed as a discrete controller for reproduction. Our approach was validated through an experiment in which a robotic arm and hand play a small musical passage on a keyboard. The experiment showed that our model is able to model the task, and enforce synchronization between components in reproduction despite the presence of delays in execution. Especially on accelerated reproduction speeds, the TPN synchronized individual components significantly better compared to the baseline. While simple trajectory replay with high stiffness can result in better reproduction than TPNs obtained from single demonstrations especially at low speeds, the advantage of our approach is the acquisition of concise models from multiple demonstrations that can be combined to generalize the task. The experiment also showed that some complex tasks can in fact be modeled as a combination of simple movements of different components.

In future work, multiple demonstrations will be included to generalize the task structure as well as TPN parameters. A more sophisticated primitive framework, such as dynamic movement primitives (DMP), will be used to increase modeling efficiency and the quality of reproductions. Furthermore, trainable classifiers can be used to raise events to fire transitions based on sensory features from other concurrently active components, which should allow for more precise synchronization. We will also investigate the possibility of propagating information through the net via changing token colors based on execution history. For example, actual execution times of previous primitives can be used to modulate the execution of latter ones. Lastly, hierarchical TPNs can be used to reduce the complexity of learning more complex tasks, in which places in a high-level net can denote actions and can be expanded into low-level nets to describe these actions in term of primitives.

ACKNOWLEDGMENT

The work leading to these results has received financial support from Academy of Finland (grant No. 264239).

REFERENCES


TABLE I

<table>
<thead>
<tr>
<th>Normal Speed</th>
<th>Double Speed</th>
<th>Triple Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPN</td>
<td>Repl.</td>
<td>TPN</td>
</tr>
<tr>
<td>$\mu_{ITE}$</td>
<td>57.5</td>
<td>40.9</td>
</tr>
<tr>
<td>$\sigma_{ITE}$</td>
<td>56.2</td>
<td>25.2</td>
</tr>
<tr>
<td>$\mu_{ED}$</td>
<td>15.6</td>
<td>-</td>
</tr>
<tr>
<td>$\sigma_{ED}$</td>
<td>4.1</td>
<td>-</td>
</tr>
<tr>
<td>Hand $\mu_{ED}$</td>
<td>-6.2</td>
<td>-</td>
</tr>
<tr>
<td>Hand $\sigma_{ED}$</td>
<td>10</td>
<td>-</td>
</tr>
</tbody>
</table>