Classification-based Parameter Synthesis for Parametric Timed Automata

Jiaying Li[1], Jun Sun[1], Bo Gao[1] and Étienne André[2]

[1] Singapore University of Technology and Design
[2] LIPN, University Paris 13
jiaying_li@mymail.sutd.edu.sg, {sunjun,bo_gao}@sutd.edu.sg,
Etienne.Andre@univ-paris13.fr

Abstract. Parametric timed automata are designed to model timed systems with unknown parameters, often representing design uncertainties of external environments. In order to design a robust system, it is crucial to synthesize constraints on the parameters, which guarantee the system behaves according to certain properties. Existing approaches suffer from scalability issues. In this work, we propose to enhance existing approaches through classification-based learning. We sample multiple concrete values for parameters and model check the corresponding non-parametric models. Based on the checking results, we form conjectures on the constraint through classification techniques, which can be subsequently confirmed by existing model checkers for parametric timed automata. In order to limit the number of model checker invocations, we actively identify informative parameter values so as to help the classification converge quickly. We have implemented a prototype and evaluated our idea on 24 benchmark systems. The result shows our approach can synthesize parameter constraints effectively and thus improve parametric verification.

1 Introduction

Timed-automata [2] are finite-state automata extended with real-valued clock variables which capture the passage of time. As a modeling language, timed-automata are used to model embedded software, timed protocols, cyber-physical systems, etc. To verify such systems, a number of verifiers on timed automata have been developed [10,21,38,40], including the well-known UPPAAL [10] model checker, which has been applied to several industrial applications [39].

In timed automata, clock variables are compared with concrete constants within clocks guards. However, these constants may be unknown at the design time. If an embedded software interacts with an external environment, the constants may depend on the environment. Furthermore, the use of parameters is fundamental in the early phases of the development, giving the possibility to explore different design choices [13]. For example, "given a real-time system M with unknown constants d and r, representing the deadline and the delay in receiving an acknowledgment, one may wish to verify a property F of the system." [24]. To design such a system robustly, it may be useful to have a timed automaton model where r and d are kept as unknown parameters, since concrete values for them make sence only in a given concrete environment.

Therefore, parametric timed automata (PTA [3]) which extend timed automata with parametric clock guards have been proposed. The concrete behavior of a PTA depends on the valuation of its parameters, and therefore a given property can be verified for some valuations only in general. A main goal of system verification will be to synthesize a set of valuations (often in the form of a convex or non-convex constraint) for which a PTA satisfies a property; this is also a way to explore various design choices at once. However, manual estimation is time-consuming and does not always generate optimal solutions for specific design problems. In contrast, parametric model checking (i.e., model checking of parametric models [23,25,26,7]) aims to automatically synthesize the region of property-satisfying parameter values, in the form of a constraint.

Existing work on the PTA verification problem relies on exploring PTA models and synthesizing constraints based on "bad states" or "good states". Given a set of property-violating states (hence "bad") or property-satisfying states (hence "good"), we can synthesize a sound constraint by either covering all the "good states" or avoiding all the "bad states". For instance, [16] proposes a method based on the counterexample-guided abstraction refinement (CEGAR). Firstly, the PTA is explored through model checking where parameters are kept as a part of the symbolic state space. After finding a counterexample, a constraint which makes the counterexample infeasible is identified. Afterwards, a different counterexample is identified and subsequently a different constraint. Once all the counterexamples are eliminated, the disjunction of these identified constraints captures all the property-satisfying parameter values.

Note that these approaches often suffer from scalability problem, which limits their power in practice. For example, IMITATOR [5] times out when applied to check a parametric Fischer protocol with 5 processes. In comparison, UPPAAL can verify the non-paremtric Fischer protocol with dozens of processes [9]. Furthermore, existing approaches provide no information if they fail to handle a given model.

In this work, we propose an approach to enhance the scalability of existing model checkers for PTA by adopting machine learning techniques. The idea is to form conjectures on the constraint based on sampling and classification techniques. Our approach takes a PTA as input and works as follows. Firstly, we generate random parameter values and construct the corresponding non-parametric timed automata. Next, we verify the timed automata using existing model checker (i.e., UPPAAL). Based on the checking results, we form conjectures on the constraint through machine learning, which can be subsequently checked using existing model checkers for PTA (i.e., IMITATOR). Moreover, we actively seek out informative parameter values and check the corresponding timed automata so that we converge to an accurate conjecture quickly. We implement our approach as a tool called PTA-LEARN and evaluate it on benchmark systems. We also compare it with state-of-the-art tools such as IMITATOR [5]. The results show our approach can synthesize parameter constraints effectively and thus improve parametric verification. Since machine learning algorithms used in our approach are agnostic with the underlying system and learn only based on the verification results of the non-parametric timed automata, our approach is more scalable than the existing approaches.

The remainders of the paper are organized as follows. Section 2 introduces a simple protocol and then illustrates how our approach works step-by-step. Then, Section 3 shows how candidate constraints are generated through classification and refined

through active learning. Next, Section 4 evaluates our approach using a set of benchmark models. Section 5 reviews related work and Section 6 concludes in the end.

2 The Overall Approach

In this section, we first define the parametric model checking problem of timed automata and then illustrate how our approach works on an example system. We start with defining our model, i.e., timed automata and parametric timed automata.

2.1 Problem Definition

Let $\mathbb{R}^{\geq 0}$ be the set of non-negative real numbers. Given a set of clocks X, we define $\Phi(C)$ as the set of clock constraints. Each clock constraint is inductively defined by: $\delta := true \mid x \sim n \mid \delta_1 \wedge \delta_2 \mid \neg\delta_1$ where $\sim \in \{=, \leq, \geq, <, >\}$; x is a clock in X and $n \in \mathbb{N}^{\geq 0}$ is a constant. The set of downward constraints obtained with $\sim \in \{\leq, <\}$ is denoted as $\Phi_{\leq,<}(X)$. A clock valuation v for a set of clocks X is a function which assigns a real value to each clock. A clock constraint can be viewed as the set of clock valuations which satisfy the constraint. A clock valuation v satisfies a clock constraint δ, written as $v \in \delta$, iff δ evaluates to be true using the clock values given by v.

Definition 1. *A timed automaton is a tuple $\mathcal{A} = (S, Init, \Sigma, X, L, T)$ where S is a finite set of locations; $Init \subseteq S$ is a set of initial locations; Σ is an alphabet; X is a finite set of clocks; $L : S \to \Phi_{\leq,<}(X)$ labels each state with an invariant; $T \subseteq S \times \Sigma \times \Phi(X) \times 2^{|X|} \times S$ is a labelled transition relation.*

Intuitively, a transition $(s, e, \delta, \chi, s') \in T$ can be fired if δ is satisfied. After event e occurs, clocks in χ are set to zero. The concrete semantics of $\mathcal{A}$ is an infinite-state labelled transition system (LTS), denoted as $\mathcal{C}(\mathcal{A}) = (S_x, Init_x, \mathbb{R}^{\geq 0} \times \Sigma, T_x)$ such that S_x is a set of concrete states of $\mathcal{A}$, each of which is a pair (s, v) where $s \in S$ is a state and v is a clock valuation; $Init_x = \{(s, X = 0) \mid s \in Init\}$ is a set of initial concrete states; and T_x is a set of concrete transitions of the form $((s, v), (d, e), (s', v'))$ such that there exists a transition $(s, e, \delta, \chi, s') \in T$; $v + d \in \delta$; $v + d \in L(s)$; $[\chi \mapsto 0](v + d) = v'$; and $v' \in L(s')$. Intuitively, the system idles for d time units at state s and then take the transition (generating event e) to reach state s'.

Given a property, the model checking problem of timed automata is to model check whether the given timed automaton satisfies the property. We skip the details on how to model check timed automata and refer the readers to [43] for details.

By generalizing the timed automata theory [2], Alur et al. first defined parametric timed automata in [3], where guards and state invariants are allowed to be parametric. Let $P = \{p_1, \cdots, p_M\}$ be a set of parameters. Throughout this paper, we assume parameters are integer-valued. Let $\Phi(X, P)$ be the set of parametric clock constraints which are inductively defined by: $\gamma := \delta \mid x \sim \alpha \mid \gamma_1 \wedge \gamma_2 \mid \neg\gamma_1$ where $\delta \in \Phi(C)$ is a non-parametric constraint; $\sim \in \{=, \leq, \geq, <, >\}$; and α is a parametric linear term in the form of $\Sigma_i a_i * p_i + d$ where both a_i and d are integer constants. The set of downward parametric constraints obtained with $\sim \in \{\leq, <\}$ is denoted as $\Phi_{\leq,<}(X, P)$.

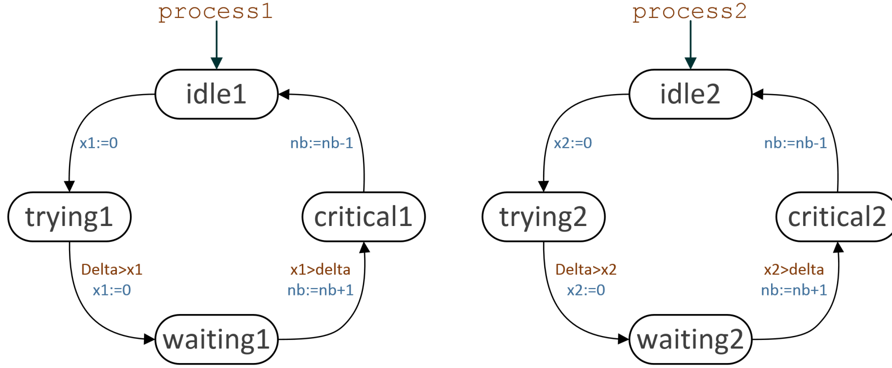

Fig. 1: Fischer protocol with 2 processes

Definition 2. *A PTA $\mathcal{A}(P)$ with parameters P is a 7-tuple $(S, Init, \Sigma, X, \phi, L, T)$ where S, $Init$, Σ, and X are the same in the timed automata definition; and*

- *$\phi \in \Phi(X, P)$ is a constraint on the parameters P;*
- *L is the invariant assigning to every $q \in S$ a constraint $L(q) \in \Phi_{\leq,<}(X, P)$ on the clocks and the parameters;*
- *and $T \subseteq S \times \Sigma \times \Phi(X, P) \times 2^{|X|} \times S$ is a labelled transition relation.*

Both timed automata and PTA can be composed in parallel. The parallel composition of two timed automata or PTA is defined in the standard way (refer to [2]). Figure 1 shows two example PTA, and the overall system is defined as their parallel composition. In this example, we use discrete integer-valued shared variables (e.g., nb), supported by most model checkers (such as UPPAAL and IMITATOR). When bounded, these variables do not add expressiveness, but act as syntactic sugar for extra locations.

Given a PTA $\mathcal{A}(P)$ and a parameter valuation v, we can construct the corresponding timed automata, written as $\mathcal{A}(v)$, by substituting the parameter values in the parameter constraints with v. Given a PTA $\mathcal{A}(P)$ and a property ρ, the parametric model checking problem is to synthesize a constraint π such that for any parameter valuation v, $\mathcal{A}(v)$ satisfies ρ if and only if v satisfies π. In particular, we say that π is sound with respect to ρ if $\mathcal{A}(v)$ satisfies ρ for all $v \in \pi$; we say that π is complete with respect to ρ if $v \in \pi$ as long as $\mathcal{A}(v)$ satisfies ρ; and we say π is perfect if it is both sound and complete. We remark existing approaches often focus on identifying sound constraints, since identifying perfect constraints are often infeasible.

2.2 Overall Approach with an Illustrative Example

In the following, we illustrate how our approach works through a simple example. We fix a PTA $\mathcal{A}(P) = (S, Init, \Sigma, X, \phi, L, T)$ in the following.

Example 1. The Fischer's protocol is a mutual exclusion protocol proposed by Fischer [6]. Instead of using atomic test-and-set instructions or semaphores, it only assumes atomic reads and writes to a shared variable and achieves mutual exclusion between multiple processes by carefully placing bounds on the execution times of the

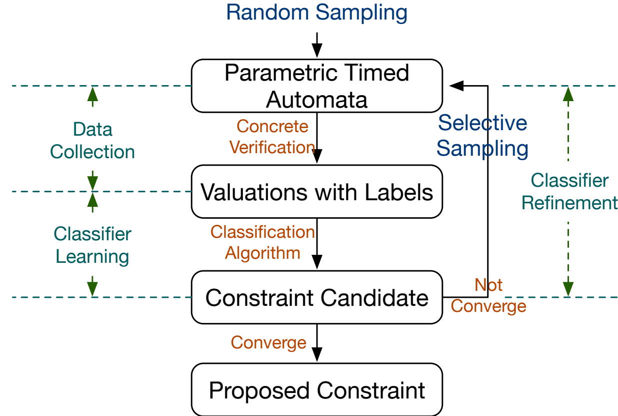

Fig. 2: Approach overview

instructions. For simplicity, we focus on the Fischer protocol with only two processes, which are modeled as the PTA in Figure 1. Each child PTA models a process with a set of four locations, with one initial location at the top and contains one clock (i.e., $x1$ for process 1 and $x2$ for process 2). The parallel composition of the two processes forms the system model. Variable nb is a shared global variable which intuitively records the number of processes in the critical session. The protocol is designed for mutual exclusion, i.e., $\Box(nb \leq 1)$ which means no more than one process should be in the critical session at any time. There are two parameters: $delta$ and $Delta$, which are used as bounds for the clocks. We remark in the original protocol [6], the property has been proved under the occasion that $delta$ is set to be 4 and$Delta$ is set to be 3. The goal of parametric model checking for this example is to find out a constraint which contains all the property-satisfying properties. For instance, one possible constraint is $delta > Delta$. In the following, we show how we can synthesize such a constraint automatically.

The overall work flow of our approach is shown in Figure 2. Given a PTA $\mathcal{A}(P)$, we start with generating a set of random valuations for P, denoted as S, which satisfy ϕ. Hereafter, we refer to the valuations in S random samples and the process of generating them "random sampling". Random sampling provides us an initial set of samples to learn the very first candidate constraint. In this work, we generate random values for each parameter in P based on its domain, assuming a uniform probabilistic distribution over all values in its domain. With each parameter valuation $v \in S$, we can generate a timed automaton model $\mathcal{A}(v)$. Next, we employ the UPPAAL to check whether $\mathcal{A}(v)$ satisfies the property. Depending on the verification results, we partition S into two sets P_S and N_S, where P_S contains all those valuations $v \in S$ such that $\mathcal{A}(v)$ satisfies the property and N_S contains all those valuations $v \in S$ such that $\mathcal{A}(v)$ fails the property.

Example 2. Continuing Example 1, assume that we generate four parameter valuations: $(6, 7)$, $(8, 2)$, $(5, 3)$ and $(0, 4)$ where each pair (d_1, d_2) denotes a valuation $\{delta \mapsto d_1, Delta \mapsto d_2\}$. Based on UPPAAL's verification results, these valuations are divided into two sets: P_S containing $\{(8, 2), (5, 3)\}$ and N_S containing $\{(6, 7), (0, 4)\}$.

Recall that the goal of parametric model checking is to synthesize a constraint which all parameter valuations in P_S should satisfy and all parameter valuations in N_S should not satisfy. We thus employ classification techniques, which have been extensively studied in machine learning community, to generate classifiers which can be treated as constraint candidates. Among the classification algorithms, e.g., [29,31,11]. we focus on two particular classification algorithms: Support Vector Machine (SVM [11]) and Kernel Query By Committee (KQBC [18]), which are introduced in Section 3.

Example 3. Continuing Example 2, assume that we apply SVM to generate a classifier to divide sets P_S and N_S. By tuning parameters in SVM, we can obtain a model as the classifier which make zero prediction error on the training set P_S and N_S. Converting the model into an explicit hyperplane, we learn the classifier as $delta - 2 * Delta \geq -4$.

Compared with the desired constraint mentioned in Example 1, this classifier is quite different. Although this desired constraint remains unknown in practice, the problem of classification on limited random samples is real. One way to solve this problem is to generate more random samples. In general, it is likely that a better constraint can be learned if more samples are provided. In our setting, it is expensive since we need to model check $\mathcal{A}(c)$ for each valuation c in order to categorize it. So it would be good if we are able to learn accurate constraints with a small number of samples.

Our remedy is to apply active learning techniques to select the most informative parameter valuations so that we converge fast. Which samples are considered most informative can be defined in different ways, depending the classification algorithms, which are detailed in Section 3. With these new samples and their corresponding labels, a new classifier can be learned. We iteratively adopt this learning and refining procedures until the generated constraint stays the same, in other words, converges. Then we can stop and report this constraint as a candidate for the constraint.

Example 4. Continuing with Example 3, given the classifier $delta - 2 * Delta \geq -4$, we pick four more valuations $(3,3)$, $(1,2)$, $(4,4)$, $(6,5)$ which locate right by the classification boundary geometrically. After verifying the corresponding timed automata using UPPAAL, the valuations $(3,3)$,$(4,4)$,$(6,5)$ are added into set P_S and $(1,2)$ is added into set N_S. Then a new classifier: $delta - Delta \geq 0$ can be obtained by classification algorithms. With the observation that the constraint converges after multiple iterations of learn-and-refine, we report this classifier as the candidate constraint.

Once having a candidate constraint $\mathcal{C}$, we employ a parametric model checker (i.e., the state-of-the-art IMITATOR [5]) for checking the correctness of ρ. That is, we construct a new PTA $\mathcal{A}'(P) = (S, Init, \Sigma, X, \mathcal{C}, L, T)$ where ϕ is replaced by $\mathcal{C}$ and solve the parametric checking problem of $\mathcal{A}'(P)$. We remark that parametric checking $\mathcal{A}'(P)$ is often easier than $\mathcal{A}(P)$, as we show empirically in Section 4. Intuitively, this is because $\mathcal{C}$ is more restrictive than ϕ and thus IMITATOR needs to explore, symbolically, a smaller state space. However, even with the learned constraint $\mathcal{C}$, soundness and completeness may not be checked from time to time still due to the complexity in parametric model checking of timed automata. Compared to directly applying a parametric model checker to $\mathcal{A}(P)$, which provides no information at all if the model checker times out, our approach provides a conjecture $\mathcal{C}$, which could be useful for system design.

Algorithm 1: Algorithm *generate*(P_S, N_S)

```
1 while not time out do
2     let C be a constraint generated by classify(P_S, N_S);
3     if C is the same as the one obtained in the last iteration then
4         return C;
5     V = select(C);
6     for v ∈ V do
7         add v into P_S if A(v) satisfies the property;
8         add v into N_S if A(v) fails the property;
```

Example 5. Continuing Example 4, we apply IMITATOR to check the soundness and completeness of the learned constraint. For this example, IMITATOR confirms that it is both sound and complete. In fact, IMITATOR can generate the same constraint for this example if no constraint provided. However, if we increase the number of processes to 5, IMITATOR is unable to synthesize any sound constraint. On the contrary, with the learned constrraint $delta - Delta \geq 0$, IMITATOR can prove the soundness and completeness of such a system, as shown in Section 4.

3 Classification

We have discussed the overall approach in Section 2. While most of the steps are self-explanatory, details on how candidate constraints are generated and refined are centric in our approach and thus will be explained in this section. The overall algorithm for generating candidate constraints is shown in Algorithm 1. Given two sets of labelled samples P_S and N_S, we first learn a candidate constraint using function $classify(P_S, N_S)$ at line 2. If the constraint is the same as the one obtained in the last iteration, we consider that the constraint has converged and return it. Otherwise, we selectively generate a set of new parameter valuations using function $select(\mathcal{C})$ at line 5. The loop from line 6 to 8 then checks whether each parameter valuation is property-satisfying or not and adds it into either P_S or N_S depending on the verification result. The outer loop from line 1 to 8 iterates until a constraint is returned at line 4 or a timeout has occurred. In the following, we present details of function $classify(P_S, N_S)$ and $select(\mathcal{C})$.

3.1 Classification

Function $classify(P_S, N_S)$ generates a candidate constraint based on classification techniques. Assume that π is the perfect constraint for which the PTA satisfies the property. Intuitively, since parameter valuations in P_S must satisfy π (since P_S contains valuations that have been checked to satisfy π) and valuations in N_S must not satisfy π, a constraint $\mathcal{C}$ separating the two sets (a.k.a. a classifier) thus can be regarded as a candidate for π. In the extreme case, if we can enumerate all the possible parameter valuations, a classifier which perfectly separates the sets is equivalent to π.

To automatically generate classifiers separating P_S and N_S, we apply existing classification techniques. In the machine learning setting, the assumption is that there is a training set containing samples X and the associated labels Y, and the goal of classification is to learn a function $f : X \to Y$ which accurately predicts the labels of samples arising in the future. There are many existing classification algorithms. For instance, k-nearest neighbors algorithm [14] clusters samples into groups based on their distances to others, while decision tree algorithm [31] splits set of samples step by step according to the maximal information gain of the unused features. Moreover, perceptron [29], Supported Vector Machine (SVM [11]) and Kernel Query By Committee (KQBC [18]) have been proposed to construct classifiers which can separate the samples with different labels apart. In general, due to the noises in the training set, these classification algorithms prefer a function with small prediction error (rather than zero) on the training set to avoid the overfitting problem. However, in our setting, any prediction error is intolerable and thus the classification algorithms must be tuned to generate perfect classifiers. Formally, a perfect classifier π for P_S and N_S is a predicate such that $s \in \pi$ for all $s \in P_S$ and $s \notin \pi$ for all $s \in N_S$. Furthermore, in order to help system designer utilize the learned constraint, it is preferred to be human-interpretable. Considering all these mentioned above, we briefly introduce one of the classification algorithms, SVM, which we adopt in our work.

SVM is a commonly applied supervised machine learning algorithm for classification and regression analysis [11]. In the binary classification case, the functionality of SVM works as follows. Given P_S and N_S, SVM can generate a perfect classifier to separate them if there is any. We refer the readers to [30] for details on how the classifier is computed. In this work, we always choose the *optimal margin classifier* if possible. Intuitively, the optimal margin classifier could be seen as the strongest witness why P_S and N_S are different. SVM by default learns classifiers in the form of a linear inequality, i.e., a half space in the form of $c_1x_1 + c_2x_2 + \cdots \geq k$ where x_i are variables while c_i and k are constant coefficients.

As linear inequalities may not be sufficiently expressive for some parametric models, we discuss how SVM can be extended to learn more expressive constraints. A polynomial classifier can be obtained by systematically mapping the samples to a high dimensional space and then applying SVM in the high dimensional space. For instance, assume that the maximum degree of the polynomial is set to be 2, the sample valuation $\{x \mapsto 2, y \mapsto 1\}$ in P_S is mapped to $\{x \mapsto 2, y \mapsto 1, x^2 \mapsto 4, xy \mapsto 2, y^2 \mapsto 1\}$. Let P'_S and N'_S be the set of samples in the high dimensional space. SVM is then applied to learn a perfect linear classifier for P'_S and N'_S. Mathematically, a linear classifier in the high dimensional space is the same as a polynomial classifier in the original space [22].

To generate conjunctive classifiers, we adopt the algorithm proposed in [37]. The idea is to pick one sample s from N_S each time and identify a classifier $\mathcal{C}_i$ to separate P_S and $\{s\}$, remove all samples from N_S which can be correctly classified by $\mathcal{C}_i$, and then repeat the process until N_S becomes empty. The conjunction of all the classifiers $\mathcal{C}_i$ is then a perfect classifier separating P_S and N_S. We refer the readers to [37] for details of the algorithm. We remark that if we switch P_S and N_S, the negation of the learned classifier using this algorithm is a classifier which is in the form of a disjunction.

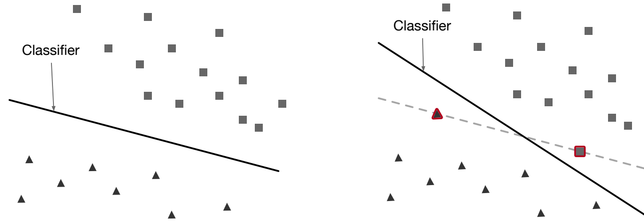

Fig. 3: Selective sampling for SVM

3.2 Candidate Refinement

Stone's celebrated theorem proves that even naive algorithms can get the optimal solution if given a large enough training sequence [18]. However, we always have obstacles in collecting such a large data set. In particular, labeling more samples is expensive in our setting because we are required to model check the system for each parameter valuation. One fundamental problem in applying classification techniques to learn the constraint is that with the limited samples in P_S and N_S, it is unlikely that we can obtain an "accurate" classifier. In the machine learning community, researchers have studied extensively on the problem "how can we learn an accurate classifier from a small number of labelled samples?". One of the remedies is active learning [34].

Active learning is proposed in contrast to passive learning. A passive learner learns from a given set of samples that it has no control over, whereas an active learner is able to adaptively select its samples. Intuitively, by selecting the right samples, active learning is able to learn much faster. In general, an active learner could choose the most informative samples to label based on the intermediate learning results. Specifically, a number of different active learning strategies on how to select the samples have been proposed. For instance, version space partitioning [32] tries to select samples on which there is maximal disagreement between classifiers in the current version space (e.g., the space of all classifiers which are consistent with the given samples); uncertainty sampling [27] maintains an explicit model of uncertainty and selects the sample that it is least confident about. The effectiveness of these strategies can be measured in terms of the labeling cost, i.e., the number of labelled samples needed in order to learn a classifier which has a classification error bounded by some threshold ϵ. An active learner can sometimes achieve good performance using far fewer samples than would otherwise be required by a passive learner [41,42]. Thus, in this work, we adopt two active learning strategies designed for different classification algorithms so that we can generate the constraint by invoking a model checker only a small number of times.

Selective Sampling for SVM We adopt the active learning strategy proposed in [33], called selective sampling, to improve the constraints generated by SVM. This strategy has been shown to be effective in different applications [41,42]. The idea is to generate multiple samples on the current classification boundary $\mathcal{C}$.

The exact details of function $select(\mathcal{C})$ in Algorithm 1 depends on the type of classifiers. For classifiers in the form of linear inequalities or polynomial inequalities, identifying samples on the classification boundary is straightforward, i.e., we turn the inequality into an equation and solve the equation. For the classifier $delta - 2 * Delta \geq -4$ in

Algorithm 2: Algorithm $kqbc_classify(\mathcal{C}, P_S, N_S)$

1. $i \leftarrow 0$;
2. **while** $i <$ ITERATION **do**
3. let $\mathcal{C}_a, \mathcal{C}_b$ be two random hypotheses selected over $\mathcal{C}$;
4. get a parameter valuation v by solving $\mathcal{C}_a(v) * \mathcal{C}_b(v) < 0$;
5. model check $\mathcal{C}(v)$;
6. add v into P_S or N_S based on whether $\mathcal{C}(v)$ satisfies the property;
7. update $\mathcal{C}$;
8. $i \leftarrow i + 1$;

the above example, we solve the equation $delta - 2 * Delta = -4$ in the integer domain and obtain new valuations like $(4, 4)$, $(6, 5)$. Note that if there is no integer solution, we solve the equation in the real-number domain and select the nearest integer samples with unknown labels. In the case that the constraint is conjunctive or disjunctive, we apply the above selective sampling approach to each clause in the constraint to obtain new samples. For instance, if $\mathcal{C}$ is in the form of $\mathcal{C}_1 \wedge \mathcal{C}_2$ where $\mathcal{C}_i$ is a linear or polynomial inequality, we turn each $\mathcal{C}_i$ into an equation and solve it to obtain new samples.

Figure 3 visualizes how selective sampling works in a 2-D plane. In the left figure, the squares represent the samples in P_S, while the triangles represent the samples in N_S. Based on these samples, a classifier is learned to separate these samples, as shown in the left figure. Selective sampling allows us to identify those samples (i.e., those triangles and squares on the line) on the classification boundary based on the learned classifier. The classifier is then improved using the new samples generated by selective sampling, as shown in the right figure.

KQBC Although SVM is a widely used classification technique and its selective sampling strategy works often in practice [41,42], it has been shown that SVM-based active learning in the worse case has the same labeling cost as random sampling, i.e., $\Omega(\frac{1}{\epsilon})$ where ϵ is the target classification error rate. A number of active learning algorithms with better worse case labeling cost have been proposed. One example is the Kernel Query By Committee (KQBC) algorithm [18]. It has been shown that KQBC has the optimal labeling cost: $O(d \lg \frac{1}{\epsilon})$ where d is the dimension of the samples [19,15]. That is, if passive learning requires a million samples, KQBC may require just $\lg 1000000$ (≈ 20) to achieve the same accuracy. Thus, in this work, we additionally adopt KQBC and develop a particular sampling strategy for KQBC to solve our problem.

Compared to SVM, instead of learning one hyperplane for separating P_S and N_S, KQBC maintains a "committee", i.e., a cluster of models $\mathcal{C} = \langle \mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3, \cdots, \mathcal{C}^m \rangle$, based on the currently labelled samples. These models compose a version space, where each member is allowed to vote on the labels of a new sample (i.e., whether a parameter valuation would make the PTA satisfy the property). KQBC shrinks the version space whenever a newly labelled sample is provided. The essence of KQBC is to constrain the size of version space as much as possible with as few labelled samples and the classification task is to search for the best model within the version space.

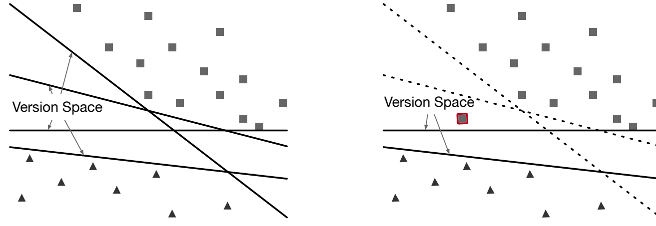

Fig. 4: Sampling in KQBC

In the original algorithm [18], KQBC takes a stream of unclassified samples and decides whether to ask for the label of a newly arrived sample. In our setting, we modify the algorithm in order to actively seek out samples which are effective in reducing the version space, and as a result we can potentially converge to the actual classifier. Algorithm 2 shows how KQBC is adopted in our setting, where the input parameter $\mathcal{C}$ represents the version space, P_S and N_S are the positive and negative samples. At line 3, we randomly pick two hypotheses (i.e., hyperplanes) $\mathcal{C}_a$ and $\mathcal{C}_b$ in the current version space $\mathcal{C}$. At line 4, we employ a constraint solver to solve the constraint $\mathcal{C}_a(v) * \mathcal{C}_b(v) < 0$ where $\mathcal{C}_a(v)$ is the label prediction of sample v, which is either 1 or -1. That is, by solving the constraint, we identify a controversial sample, i.e., one which is disagreed upon by two members of the committee. At line 5, we model check the timed automaton $\mathcal{C}(v)$ and we add v into P_S or N_S accordingly at line 6. At line 7, we update the version space. We skip the details on how the version space is updated and maintained and refer the readers to [18] for the technical details. The loop from line 2 from line 8 iterates until a pre-defined number of iterations has been reached.

Figure 4 illustrates how classfiers are obtained by KQBC in a binary classification task. In the left figure, the squares represent samples in P_S while the triangles represent samples in N_S. All the lines compose the committee for the current samples. Note that any member of the committee classifies the current samples perfectly. In order to reduce the committee, we select two lines by hit-and-run algorithm [28] and identify a sample which they disagree upon, represented as the bigger square in between the lines. After obtaining the label of this sample, two members of the committee represented by the dotted lines are ruled out. As a result, the version space is reduced.

4 Evaluation

We have implemented our approach for model checking of PTA in a tool named PTA-LEARN (available at [1]). PTA-LEARN is written using a combination of C++ and shell codes. It makes use of GSL [20] to solve equation systems; and uses LibSVM [12] for SVM-based classification. It relies on UPPAAL [10] for model checking timed automata and IMITATOR for model checking PTA with learned constraints. Note that both UPPAAL and IMITATOR are regarded as the state-of-the-art in their respective fields. In the following, we evaluate PTA-LEARN, to address the following three research questions.

- RQ1: Can PTA-LEARN improve scalability of IMITATOR?
- RQ2: Are the constraints generated by PTA-LEARN sound, or complete compared to those generated by IMITATOR?

– RQ3: Is our candidate refinement strategy helpful?

To answer the aforementioned research questions, we identify 24 parametric timed automata models from the IMITATOR benchmarks library, which in terms are collected from multiple sources. Since the models in [5] are written in a language different from the language supported by UPPAAL, we develop a translator to convert those models. The correctness of the translator is checked manually as well as through comparing verification results of UPPAAL and IMITATOR. All evaluated models are available at [1].

The parameters in our experiments are configured as follows. During the random sampling stage, given a parametric model $\mathcal{A}(P)$ and a property ρ, we try to generate random parameter valuations until there are at least one valuation v such that $\mathcal{A}(v)$ satisfies ρ and one valuation v' such that $\mathcal{A}(v')$ fails ρ. If all random valuations satisfy ρ after a threshold $64 * |P|$ of random values (where $|P|$ is the number of parameters), we stop the process and conjecture that the constraint is $true$ (i.e., any parameter valuation is valid). Similarly, if all random valuations fail ρ, we conjecture that the constraint is $false$ (i.e., no parameter valuation is valid).

During the learning stage, both SVM and KQBC are applied and we take the first converged constraint as the learning result. In the case of SVM, the parameter C (which controls the trade-off between avoiding misclassifying training samples and enlarging the decision boundary) in LibSVM and the inner iteration are set to their maximum values so that it generates only the perfect classifier, if there is. Selective sampling is applied repeatedly until the learned constraint remained unchanged after 2 consecutive iterations. We remark it is an open question on how to know that a classifier has converged. In the case of KQBC, we conduct a set of preliminary experiments with randomly generated predicate to test how many samples are necessary to learn the predicate. The details of the preliminary results are available at [1]. During the verification stage, if IMITATOR is applied, the timeout is set to be 300 seconds. If PTA-LEARN learns a constraint $true$ or does not learn (i.e., timeout), we apply IMITATOR on the original model. Each experiment is repeated for 5 times and we report the median as the experiment results. All of the experiments are conducted using x64 Ubuntu 16.04.2 (kernel 4.8.0-49-generic) with 3.60 GHz Intel Core i7 and 32G DDR3.

The experiment results are shown in Table 1. To answer RQ1, we apply PTA-LEARN to each model and compare the performance with IMITATOR. The result verification time of IMITATOR is shown in the second column of Table 1. The following two columns show the time spent on learning and verifying the constraint by PTA-LEARN. PTA-LEARN*'s total time is the sum of numbers in these two columns.* It can be observed that IMITATOR times out in 9 cases. In comparison, PTA-LEARN succeeds in learning constraints for 21 out of 24 benchmarks and fails to learn any constraint for the 3 remaining case studies. A close look reveals that in the 3 cases, 2 models involve many parameters (i.e., > 6) and thus our learning algorithms time out before converge. Note that IMITATOR times out on these two cases as well. We fail to learn on the other case because the actual constraint is a complicated constraint consisting of a mixture of conjunctions and disjunctions. Recall that if we do not learn any constraint, the original model is submitted to IMITATOR for parametric model checking. It can be observed that (as shown in the column "verify" under PTA-LEARN) that with the learned constraint, PTA-LEARN is able to verify 23 models and only times out on one model

Model	IMITATOR	PTA-LEARN				PTA-LEARN-Active		
		learn	verify	sound	complete	learn	verify	sound
coffee	**0.007**	1.482	0.017	T	T	1.851	0.017	T
coffeeDrinker	**0.019**	timeout	0.019	-	-	timeout	0.019	-
counterexACSD15	timeout	**1.49**	**0.018**	T	T	1.128	timeout	-
ex1pPTA	timeout	**3.134**	**0.030**	T	-	20.34	0.018	T
exUPTA-allp	timeout	**1.148**	**0.018**	T	-	0.375	timeout	-
F3	**0.184**	0.171	0.014	T	T	0.175	0.014	T
F4	28.777	**0.763**	**4.151**	T	T	1.052	4.095	T
F5	timeout	**0.851**	**227**	T	-	1.073	243	T
FischerAHV93	**0.040**	7.835	0.039	T	T	12.56	0.405	F
fischerHRSV02-2	timeout	**14.686**	**0***	T	-	10.88	timeout	-
fischerHRSV02-3	timeout	**7.670**	**0***	T	-	7.22	timeout	-
fisherPAT.nocomment	**0.05**	2.886	0.036	T	T	2.917	0.0367	T
IMPO	**0.013**	timeout	0.013	-	-	timeout	0.013	-
JLR13-3tasks-npfp	36.84	**1.91**	**0***	T	F	0.8	timeout	-
JLR-TACAS13	timeout	**0.694**	**0.016**	T	-	0.688	0.015	T
LALSD14-FMS2p	timeout	6.142	timeout	-	-	8.65	timeout	-
NuclearPlant	**0.023**	4.47	0.017	T	T	3.196	0.022	F
Pipeline-KP12-2-5	**13.323**	timeout	13.323	-	-	timeout	13.323	-
Sched2.100.0	**1.11**	3.84	0.318	T	F	1.654	1.347	F
Sched2.50.0	**0.933**	2.782	0.302	T	T	1.278	1.177	F
testBadWithoutDiscrete	**0.019**	1.212	0.018	T	T	0.347	0.018	T
testIM-IMK-IMunion	**0.008**	0.305	0.016	T	T	0.4	0.016	T
TestPattern1	**0.018**	2.95	0.031	F	-	2.28	0.035	F
WFAS-BBLS15-det	timeout	**1.925**	**0.027**	T	-	1.216	0.221	F

Table 1: Evaluation results, where “-” means “not applicable”

named LALSD14-FMS2p. In terms of efficiency, we highlight the approach between PTA-LEARN and IMITATOR which takes less time in parametric verification for each model. PTA-LEARN takes seconds on learning and verifies the models more efficiently with the learned constraint, than applying IMITATOR directly. Thus, we conclude that PTA-LEARN can be used to improve IMITATOR.

To answer RQ2, we measure the soundness and completeness of the learned constraint. Soundness of a learned constraint is checked by IMITATOR, i.e., we apply IMITATOR to check whether the PTA updated with the learned constraint always satisfies the property. We recall that a sound constraint is useful as it provides a guideline for choosing safe parameter values. Column $sound$ under PTA-LEARN shows whether the learned constraint can satisfy the properties. The results show that, among the 21 constraints learned by PTA-LEARN, 19 are proved sound; 1 is not sound; and 1 is unknown as PTA-LEARN times out trying to prove its soundness.

A sound constraint may be too restrictive. In the extreme case, the constraint $false$ is trivially sound and obviously not useful. Out of the 24 cases, in three cases, we learn the trivial constraint $false$, marked as “*” in Table 1. To checking the completeness of the learned constraint, we compare the learned constraint with the constrained obtained by applying IMITATOR directly. We consider the learned constraint is complete if and only if it is weaker than or equivalent to that obtained by IMITATOR. The results are

shown in column *complete*. It is observed that out of the 12 constraints for which we can evaluate the completeness, 10 of them are complete. For the remaining two cases, IMITATOR finds constraints which are weaker than those found by PTA-LEARN. This is possible as PTA-LEARN is based on black-box learning, whereas as a white-box technique, IMITATOR explores the system paths systematically if it is able to finish.

To answer RQ3, we compare the performance of PTA-LEARN with and without active learning. In our evaluation, when active learning is not applied, we simply learn from randomly generated parameter valuations. That is, we keep generating random valuations until the constraints get converged. The columns under **PTA-LEARN**-active show the results obtained by applying PTA-LEARN without active learning. Note that the number of sound constraints reduces from 19 to 9 when active learning is disabled. One reason is that without active learning, often different runs of the same experiment result in different constraints, which makes it hard for the constraints to converge. Comparing PTA-LEARN with and without active learning, we can see that the overhead of active learning is negligible. We thus conclude that active learning is helpful.

5 Related Work

Basides our method, several white-box tools have been developed to verify parametric systems by exploring system states with different strategies. For instance, LPMC [38] employs a partition refinement technique to generate an unstructured set of constraints; HYTECH [21] verifies linear hybrid automata by exploring the state space through either forward reachability or partition refinement; [23] adopts a symbolic representation of the state space to synthesize linear parameter constraints. IMITATOR [5] implements the inverse method (or trace preservation synthesis), the behavioral cartography [4], bad state reachability synthesis (used in this work), parametric deadlock-freeness checking and non-Zeno parametric model checking. The idea of behavioral cartography is close to our sampling that iterates on integer-valuations to generalize their discrete behavior; however, a main difference (and advantage) of our work is that we use non-parametric model checking on the sampled points, which is more efficient by an order magnitude, and we only use parametric model checking to assess the validity of the constraint.

To the best of our knowledge, we are the first to introduce learning techniques in verifying parametic systems. But this is not a genius creation, as learning has been already applied in many other areas successfully. For instance, there are many learning-based approaches in program verification field. In particular, several papers [37,36,35,17] have deployed learning techniques to help software verification, and compiler optimization. In these works, program states are regarded as labelled samples and a variety of classification algorithms are applied to learn the relationship between a correct program with the program states. We remark that although we focus on PTA throughout this paper, our technique can be adapted to other models like parametric probabilistic models [8].

6 Conclusion

In this work, we propose an approach to automatically synthesize parameter constraints through learning. In particular, we apply active learning techniques so as to learn accurate candidate constraints prior to the checking phase. Furthermore, we adopt SVM

and KQBC as the classification algorithms to learn constraints in different forms. In principle, our approach can be extended to learn arbitrary mathematical classifiers with kernel methods. Nonetheless, we focus on constraints in form of polynomial inequalities or conjunctions/disjunctions of polynomial inequalities in our evaluation. The results show that our approach effectively learns parameter constraints to guarantee the correctness of a set of benchmarks and hence helps the system verification and design.

7 Acknowledgement

This work is supported by NRF project "RG101NR0114A".

References

1. PTA-Learn repo. `https://github.com/lijiaying/pta-Learn`, 2017.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
3. R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *Proceedings of the 25th annual ACM symposium on Theory of Computing*, pages 592–601. ACM, 1993.
4. É. André and L. Fribourg. Behavioral cartography of timed automata. In A. Kučera and I. Potapov, editors, *RP*, pages 76–90. Springer, aug 2010.
5. É. André, L. Fribourg, U. Kühne, and R. Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In D. Giannakopoulou and D. Méry, editors, *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *Lecture Notes in Computer Science*, pages 33–36. Springer, Aug. 2012.
6. D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing population protocols. In *International Conference On Principles Of Distributed Systems*, pages 103–117, 2005.
7. L. Atefnoaei, S. Bensalem, M. Bozga, C. Cheng, and H. Ruess. Compositional parameter synthesis. In J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou, editors, *FM*, volume 9995 of *Lecture Notes in Computer Science*, pages 60–68, 2016.
8. C. Baudrit, D. Dubois, and N. Perrot. Representing parametric probabilistic models tainted with imprecision. *Fuzzy sets and systems*, 159(15):1913–1928, 2008.
9. G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi. Developing UPPAAL over 15 years. *Softw., Pract. Exper.*, 41(2):133–142, 2011.
10. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaala tool suite for automatic verification of real-time systems. *Hybrid Systems III*, pages 232–243, 1996.
11. B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *workshop on Computational learning theory*, pages 144–152. ACM, 1992.
12. C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.
13. A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. Parameter synthesis with ic3. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 165–168. IEEE, 2013.
14. T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.
15. S. Dasgupta. Coarse sample complexity bounds for active learning. In *NIPS*, pages 235–242, 2005.
16. G. Frehse, S. K. Jha, and B. H. Krogh. A counterexample-guided approach to parameter synthesis for linear hybrid automata. In *HSCC*, pages 187–200. Springer, 2008.
17. P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust framework for learning invariants. In *Computer Aided Verification*, pages 69–87. Springer, 2014.

18. R. Gilad-Bachrach, A. Navot, and N. Tishby. Kernel query by committee (kqbc). Technical report, Technical Report 2003-88, Leibniz Center, the Hebrew University, 2003.
19. R. Gilad-Bachrach, A. Navot, and N. Tishby. Query by committee made real. In *NIPS*, pages 443–450, 2005.
20. B. Gough. *GNU scientific library reference manual*. Network Theory Ltd., 2009.
21. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. In *International Conference on Computer Aided Verification*, pages 460–463. Springer, 1997.
22. T.-M. Huang, V. Kecman, and I. Kopriva. *Kernel based algorithms for mining huge data sets*, volume 1. Springer, 2006.
23. T. Hune, J. Romijn, M. Stoelinga, and F. W. Vaandrager. Linear parametric model checking of timed automata. *Journal of Logic and Algebraic Programming*, 52-53:183–220, 2002.
24. F. Jahanian. Verifying properties of systems with variable timing constraints. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 319–328. IEEE, 1989.
25. A. Jovanović, D. Lime, and O. H. Roux. Integer parameter synthesis for timed automata. *IEEE Transactions on Software Engineering*, 41(5):445–461, 2015.
26. M. Knapik and W. Penczek. Bounded model checking for parametric timed automata. *Transactions on Petri Nets and Other Models of Concurrency*, 5:141–159, 2012.
27. D. D. Lewis and W. A. Gale. A sequential algorithm for training text classifiers. In *SIGIR Forum*, pages 3–12, 1994.
28. L. Lovász and S. Vempala. Hit-and-run is fast and fun. *preprint, Microsoft Research*, 2003.
29. M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry, 2nd edition*. The MIT Press, 1972.
30. J. Platt et al. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.
31. J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.
32. R. A. Ruff and T. G. Dietterich. What good are experiments? In *Proceedings of the Sixth International Workshop on Machine Learning (ML 1989)*, pages 109–112, 1989.
33. G. Schohn and D. Cohn. Less is more: Active learning with support vector machines. In *ICML*, pages 839–846, 2000.
34. B. Settles. *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
35. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification*, pages 88–105. Springer, 2014.
36. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *Static Analysis Symposium*, pages 388–411, 2013.
37. R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *Computer Aided Verification*, pages 71–87. Springer, 2012.
38. R. Spelberg, H. Toetenel, and M. Ammerlaan. Partition refinement in real-time model checking. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 143–157. Springer, 1998.
39. M. Stoelinga. Fun with firewire: A comparative study of formal verification methods applied to the ieee 1394 root contention protocol. *Formal Aspects of Comp.*, 14(3):328–337, 2003.
40. J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In *International Conference on Computer Aided Verification*, pages 709–714. Springer, 2009.
41. S. Tong and E. Y. Chang. Support vector machine active learning for image retrieval. In *Proceedings of the 9th ACM International Conference on Multimedia*, pages 107–118, 2001.
42. S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.
43. S. Yovine. Model checking timed automata. *Lectures on Embedded Systems*, pages 114–152, 1998.