Runtime Verification Prediction for Traces with Data

Moran Omer and Doron Peled$^{(\boxtimes)}$

Department of Computer Science, Bar Ilan University, Ramat Gan, Israel
doron.peled@gmail.com

Abstract. Runtime verification (RV) can be used for checking the execution of a system against a formal specification. First-order temporal logic allows expressing constraints on the order of occurrence of events and the data that they carry. We present an algorithm for predicting possible verdicts, within (some parametric) *k* events, for online monitoring executions with data against a specification written in past first-order temporal logic. Such early prediction can allow preventive actions to be taken as soon as possible. Predicting verdicts involves checking *multiple* possibilities for extensions of the monitored execution. The calculations involved in providing the prediction intensify the problem of keeping up with the speed of occurring events, hence rejecting the naive brute-force solution that is based on exhaustively checking all the extensions of a certain length. Our method is based on generating *representatives* for the possible extension, which guarantee that no potential verdict is missed. In particular, we take advantage of using BDD representation, which allows efficient construction and representation of such classes. The method is implemented as an extension of the RV tool DejaVu.

1 Introduction

Runtime verification (RV) allows verifying system executions against a specification, either online as the traces are generated or offline. Monitoring is often confined to *safety properties*, where a *failure* to satisfy the specification occurs when the inspected prefix of execution cannot be extended in any way that would satisfy the specification. The specification is typically expressed using automata or temporal logic. In particular, past time propositional temporal logic can be used to express safety properties [24], allowing an efficient RV monitoring algorithm [18]. A monitored execution trace may further consist of events that contain observed data values. To deal with observations with data, RV monitoring was extended to use first-order past temporal logic [5,15]. Other RV systems that monitor sequences of events with data include [1–4,6,11–14,16,19,25,26].

While detecting failures at run time can be used to terminate bad executions, predicting the possibility of failures before they occur can be used to employ preventing measures. We present here an algorithm for predicting a potential failure during the RV monitoring a few steps before it can potentially happen. Our prediction algorithm involves the computation of possible futures of the next *k* events.

The research performed by the authors was partially funded by Israeli Science Foundation grant 1464/18: "Efficient Runtime Verification for Systems with Lots of Data and its Applications".

P. Katsaros and L. Nenzi (Eds.): RV 2023, LNCS 14245, pp. 148–167, 2023.
https://doi.org/10.1007/978-3-031-44267-4_8

In online RV, it is essential to keep the *incremental complexity*, i.e., the amount of computation performed between consecutively events, as small as possible so that we keep up with the speed of reported events. This pace can be *smoothened-up* to a certain extent with the help of a buffer, but not for the long run. For predictive RV, the problem of minimizing the incremental complexity intensifies, as it involves analyzing different possibilities for the following k events. A straightforward algorithm goes through *all* possible event sequences for the next k steps, stopping when a failure occurs. This type of "brute-force" approach is immediately disqualified because of the incremental time complexity of $O(n^k)$, where n is the number of available possibilities for each event. In the case of monitoring events with data, n can be huge, or even, in principle, unbounded.

Our approach is based on using equivalence classes between data values that occur within events, which generate isomorphic extensions to the current observed trace. Then our algorithm restricts itself to using *representatives* from these equivalence classes for extending the current observed trace. This is shown to be sufficient to preserve the correctness of the prediction. In particular, we show how to take advantage of BDD representation, as is used in the DejaVu system [15] to calculate the equivalence classes and select representatives. We describe the algorithm and its implementation as an extension of the DejaVu system. We demonstrate the algorithm with experimental results and show that our method provides a substantial improvement over the straightforward prediction algorithm.

An early verdict for a finite trace against a propositional temporal specification, based on the agreement between all of its possible infinite extensions, can be calculated based on translating the specification into an automaton [21]. We show that for first-order properties of the form $\Box\varphi$, where φ is a past first-order temporal logic property, calculating such a verdict is undecidable. This further motivates our k-step predictive algorithm as a practical compromise, when an early prediction of failures is required. This also gives an explanation of the reason why systems like DejaVu [15] and MonPoly [5] provide only the immediate *true*/*false* verdict per each input prefix against the past first-order LTL specification φ rather than for $\Box\varphi$.

Predictive Runtime Verification (PRV), has been proposed as an extension to standard runtime verification for propositional LTL in [27,28]. There, extensions to the currently observed trace are proposed based on static analysis or abstraction of the monitored system. A prediction of runtime verdicts based on *assumptions* about the monitored system is described in [10]. This is done using SMT-based model checking. That work also performs the prediction for a first-order LTL, but this version of the logic is restricted not to have quantifiers. This approach is orthogonal to ours, where our approach does not assume any further knowledge that can be used in generating such extensions; but combining the two approaches, when possible, can be beneficial. Predictive semantics for propositional LTL was used in [28] based on providing an early verdict for *satisfaction* of all extensions or *failure* to satisfy of all extensions for an LTL property based a on minimally observed trace. Providing such verdicts is also related to the notion of *monitorability* [7], classifying a finite trace based on all of its possible extensions as *good* or *bad* respectively. An algorithm for providing such an early verdict was given in [21].

For a past time LTL φ, one can employ an efficient algorithm that returns a *true*/*false* answer per each finite prefix that is monitored. Hence, the outcome can alternate between these two results. A *false* answer for a past property φ is sufficient to provide a *failure* verdict for the safety specification φ, albeit using an automata based algorithm such as [21] could have sometimes predict that *failure* is unavoidable after a shorter prefix. In [20], *anticipatory monitoring* is defined to provide the possible future verdicts after a given trace, which is also the goal of our paper. Anticipatory monitoring allows providing further information: the shortest distance to a *true* output and the longest distance to a *false* output. That work also includes a decision procedure for calculating this information for past LTL, based on a DFS on an automaton that is used to perform the monitoring. Our goal is to provide predictions for the future verdicts for traces with data with respect to a specification in first-order past temporal logic. We use here the unifying term *predictive* monitoring to refer both to the case that we calculate the possible verdicts after a bounded number of look-ahead steps as in anticipatory monitoring, for which we provide an algorithm for first-order past temporal logic, and to the case where early verdicts based on all the possible infinite extensions are sought (for an impossibility result).

2 Preliminaries

2.1 Past Time First-Order Temporal Logic

The QTL logic, used by the DejaVu tool [15,31] and as a core subset of the logic used by the MonPoly tool [5], is a specification formalism that allows expressing properties of executions that include data. The restriction to past time allows interpreting the formulas on finite traces.

Syntax. The formulas of the QTL logic are defined using the following grammar, where *p* stands for a *predicate* symbol, *a* is a *constant* and *x* is a *variable*.

For simplicity of the presentation, we define here the QTL logic with unary predicates, but this is not due to a principal restriction, and in fact QTL supports predicates over multiple arguments, including zero arguments, corresponding to propositions. The DejaVu system, as well as the method presented in this paper and its implementation [30], fully supports predicates over multiple arguments.

$$\varphi ::= \mathit{true} \mid p(\mathrm{a}) \mid p(x) \mid (\varphi \wedge \psi) \mid \neg\varphi \mid (\varphi\, \mathcal{S}\, \psi) \mid \ominus\varphi \mid \exists x\, \varphi$$

Denote by $\eta \in sub(\varphi)$ the fact that η is a subformula of φ. A QTL formula can be interpreted over multiple types (domains), e.g., natural numbers or strings. Accordingly, each variable, constant and parameter of predicate is defined over a specific type (such type declarations can appear external to the QTL formula). Type matching is enforced, e.g., for $p(a)$ ($p(x)$, respectively), the types of the parameter of *p* and of *a* (*x*, respectively) must be the same. We denote the type of a variable *x* by $type(x)$.

Propositional past time linear temporal logic is obtained by restricting the predicates to be parameterless, essentially Boolean propositions; then, no variables, constants and quantification is needed either.

Semantics. A QTL formula is interpreted over a trace (or observation), which is a finite sequence of *events*. Each event consists of a predicate symbol and parameters, e.g., $p(a)$, $q(7)$. It is assumed that the parameters belong to particular domains that are associated with (places in) the predicates. The events in a trace are separated by dots, e.g., $p(a).q(7).p(b)$. A more general semantics can allow each event to consist of a set of predicates with parameters[1]. However, this is *not* allowed in DejaVu and in the context of this paper; for predictive RV, such generalized events can increase the complexity dramatically.

QTL subformulas have the following informal meaning: $p(\mathrm{a})$ is true if the last event in the trace is $p(\mathrm{a})$. The formula $p(x)$, for some variable x, holds if x is bound to a constant a such that $p(\mathrm{a})$ is the last event in the trace. The formula $(\varphi\, \mathcal{S}\, \psi)$, which reads as φ *since* ψ, means that ψ holds in some prefix of the current trace, and for all prefixes between that one and the current trace, φ holds. The *since* operator is the past dual of the future time *until* modality. The property $\ominus\, \varphi$ means that φ is true in the trace that is obtained from the current one by omitting the last event. This is the past dual of the future time *next* modality. The formula $\exists x\, \varphi$ is true if there exists a value a such that φ is true with x bound to a. We can also define the following additional derived operators: $\mathit{false} = \neg \mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\diamondminus\, \varphi = (\mathit{true}\, \mathcal{S}\, \varphi)$ ("previously"), $\boxminus\, \varphi = \neg \diamondminus \neg\varphi$ ("always in the past" or "historically"), and $\forall x\, \varphi = \neg\exists x\, \neg\varphi$.

Formally, let $\mathit{free}(\eta)$ be the set of free (i.e., unquantified) variables of a subformula η. Let γ be an assignment to the variables $\mathit{free}(\eta)$. We denote by $\gamma[v \mapsto a]$ the assignment that differs from γ only by associating the value a to x; when γ assigns only to the variable x, we simply write $[v \mapsto a]$. Let σ be a trace of events of length $|\sigma|$ and i a natural number, where $i \leq |\sigma|$. Then $(\gamma, \sigma, i) \models \eta$ if η holds for the prefix of length i of σ with the assignment γ.

We denote by $\gamma|_{\mathit{free}(\varphi)}$ the restriction (projection) of an assignment γ to the free variables appearing in φ. Let ε be an empty assignment. In any of the following cases, $(\gamma, \sigma, i) \models \varphi$ is defined when γ is an assignment over $\mathit{free}(\varphi)$, and $i \geq 1$.

- $(\varepsilon, \sigma, i) \models \mathit{true}$.
- $(\varepsilon, \sigma, i) \models p(a)$ if $\sigma[i] = p(a)$.
- $([x \mapsto a], \sigma, i) \models p(x)$ if $\sigma[i] = p(a)$.
- $(\gamma, \sigma, i) \models (\varphi \wedge \psi)$ if $(\gamma|_{\mathit{free}(\varphi)}, \sigma, i) \models \varphi$ and $(\gamma|_{\mathit{free}(\psi)}, \sigma, i) \models \psi$.
- $(\gamma, \sigma, i) \models \neg\varphi$ if not $(\gamma, \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models (\varphi\, \mathcal{S}\, \psi)$ if for some $1 \leq j \leq i$, $(\gamma|_{\mathit{free}(\psi)}, \sigma, j) \models \psi$ and for all $j < k \leq i$, $(\gamma|_{\mathit{free}(\varphi)}, \sigma, k) \models \varphi$.
- $(\gamma, \sigma, i) \models \ominus\varphi$ if $i > 1$ and $(\gamma, \sigma, i-1) \models \varphi$.
- $(\gamma, \sigma, i) \models \exists x\, \varphi$ if there exists $a \in \mathit{type}(x)$ such that $(\gamma[x \mapsto a], \sigma, i) \models \varphi$.

Set Semantics. We define an alternative semantics that is equivalent to the standard semantics presented above, but it presents the meaning of the formulas from a different point of view: the standard semantics defines whether a subformula holds given (1) an

[1] In the generalized semantics, the condition $\sigma[i] = p(a)$ in the definition for the $p(a)$ and $p(x)$ subformulas should be replaces with $p(a) \in \sigma[i]$ and similarly in the subsequent set semantics.

assignment of values to the (free) variables appearing in the formula, (2) a trace and (3) a position in the trace. Instead, set semantics gives the *set* of assignments that satisfy the subformula given a trace and a position in it.

Set semantics allows presenting of the RV algorithm for QTL in a similar way to the RV algorithm for propositional past time temporal logic [15]. Let $I[\varphi,\sigma,i]$ be the *interpretation function* that returns a set of assignments such that $(\gamma,\sigma,i) \models \varphi$ iff $\gamma|_{free(\varphi)} \in I[\varphi,\sigma,i]$. The empty set of assignments $\emptyset$ behaves as the Boolean constant *false* and the singleton set $\{\varepsilon\}$, which contains the empty assignment ε, behaves as the Boolean constant *true*. The union $\bigcup$ and intersection $\bigcap$ operators on sets of assignments are defined, even if they are applied to non-identical sets of variables; in this case, the assignments are extended to the union of the variables. Thus intersection between two sets of assignments A_1 and A_2 is defined like database "join" operator; i.e., it consists of the assignments whose projection on the *common* variables agrees with an assignment in A_1 and with an assignment in A_2. Union is defined as the dual operator of intersection.

Let A be a set of assignments. We denote by $hide(A,x)$ (for "hiding" the variable x) the set of assignments obtained from A after removing from each assignment the mapping from x to a value. In particular, if A is a set of assignments over only the variable x, then $hide(A,x)$ is $\{\varepsilon\}$ when A is nonempty, and $\emptyset$ otherwise. $A_{free(\varphi)}$ is the set of all possible assignments of values to the variables that appear free in φ. For convenience of the set semantics definition, we add a 0 position for each sequence σ, where I returns the empty set for each formula. The set semantics is shown in the following. For all occurrences of i, it is assumed that $i \geq 1$.

- $I[\varphi,\sigma,0] = \emptyset$.
- $I[true,\sigma,i] = \{\varepsilon\}$.
- $I[p(a),\sigma,i] =$ if $\sigma[i] = p(a)$ then $\{\varepsilon\}$ else $\emptyset$.
- $I[p(x),\sigma,i] = \{[x \mapsto a] \mid \sigma[i] = p(a)\}$.
- $I[\neg\varphi,\sigma,i] = A_{free(\varphi)} \setminus I[\varphi,\sigma,i]$.
- $I[(\varphi \wedge \psi),\sigma,i] = I[\varphi,\sigma,i] \bigcap I[\psi,\sigma,i]$.
- $I[\ominus\varphi,\sigma,i] = I[\varphi,\sigma,i-1]$.
- $I[(\varphi\, \mathcal{S}\, \psi),\sigma,i] = I[\psi,\sigma,i] \bigcup (I[\varphi,\sigma,i] \bigcap I[(\varphi \mathcal{S} \psi),\sigma,i-1])$.
- $I[\exists x\, \varphi,\sigma,i] = hide(I[\varphi,\sigma,i],x)$.

2.2 Monitoring First-Order Past LTL

We review first the algorithm for monitoring first-order past LTL, implemented as part of the DejaVu tool [15]. The algorithm is based on calculating a *summary* for the current monitored trace. The summary is used, instead of storing and consulting the entire trace, for providing verdicts, and is updated when new monitored events are reported.

Consider a classical algorithm for past time propositional LTL [18]. There, the summary consists of two vectors of bits. One vector, pre, keeps the Boolean (truth) value for each subformula, based on the trace observed so far *except* the last observed event. The other vector, now, keeps the Boolean value for each subformula based on that trace *including* the last event. Given a new event e consisting of a set of propositions, which extends the monitored trace, the vector now is calculated based on the vector pre and the event e. This is summarized below:

- $\mathsf{now}(true) = True$
- $\mathsf{now}(p) = (p \in e)$
- $\mathsf{now}((\varphi \wedge \psi)) = (\mathsf{now}(\varphi) \wedge \mathsf{now}(\psi))$
- $\mathsf{now}(\neg\varphi) = \neg\mathsf{now}(\varphi)$
- $\mathsf{now}((\varphi\, \mathcal{S}\, \psi)) = (\mathsf{now}(\psi) \vee (\mathsf{now}(\varphi) \wedge \mathsf{pre}((\varphi\, \mathcal{S}\, \psi))))$.
- $\mathsf{now}(\ominus\, \varphi) = \mathsf{pre}(\varphi)$

When a new event appears, now becomes pre, and the now values are calculated according to the above cases.

The *first-order* monitoring algorithm replaces the two vectors of bits by two vectors of *assignments*: pre, for the assignments that satisfy each subformula given the monitored trace, except the last event, and now that for the assignments that satisfy the monitored trace. The updates in the first-order case replace, according to the set semantics, negation with complementations, conjunction with intersection and disjunction with union. We will describe how sets of assignments or, equivalently, relations, can be represented as BDDs. Then, complementation, intersection and union between relations correspond back to negation, conjunction and disjunction, respectively. Thus, the BDD-based algorithm for monitoring traces with data against a QTL specification which will be presented after explaining the BDD representation, will look similar to the RV algorithm for the propositional case without data.

BDD Representation. BDD representation, as used in the DejaVu tool allows an efficient implementation of RV for traces with data against first-order past temporal logic. We enumerate data values appearing in monitored events, as soon as we first see them. We represent enumerations as bit-vectors (i.e., Binary) encodings and construct the relations over this representation rather than over the data values themselves. Bit vectors are concatenated together to represent a tuple of values. The relations are then represented as BDDs [8]. BDDs were featured in model checking because of their ability to frequently achieve a highly compact representation of Boolean functions [9,23]. Extensive research of BDDs allowed implementing optimized public BDD packages, e.g., [29].

In order to deal with unbounded domains (where only a finite number of elements may appear in a given observed trace) and maintain the ability to perform complementation, unused enumerations represent the values that have not been seen yet. In fact, it is sufficient to use one enumeration representing these values per each variable of the LTL formula. We guarantee that at least one such enumeration exists by reserving for that purpose the enumeration $11\ldots11$. We present here only the basic algorithm. For versions that allow extending the number of bits used for enumerations and garbage collection of enumerations, see [17].

When an event $p(a)$ is observed in the monitored execution, matched with $p(x)$ in the monitored property, a call to the procedure $hash(a)$ checks if this is the first occurrence of the value a in an event. Then a will be assigned a new enumeration $val(a)$, which will be stored under the key a. We can use a counter, for each variable x, to count the number of different values appearing so far for x. When a new value appears, this counter is incremented and converted to a binary (bit-vector) representation. The function $\mathbf{build}(x, val(a))$ returns a BDD that represents an assignment for the bit vector x mapped to the enumeration corresponding to a.

For example, assume that the runtime-verifier sees the input events *open*(“a”), *open*(“b”) and that it encodes the argument values with 3 bits. We use x_1, x_2, and x_3 to represent the enumerations, with x_1 being the least significant bit. Assume that the value “a” gets mapped to the enumeration $x_3x_2x_1 = 000$ and that the value “b” gets mapped to the enumeration $x_3x_2x_1 = 001$. Then, the Boolean function representing the enumerations for $\{a, b\}$ is $(\neg x_2 \wedge \neg x_3)$, which returns 1 (*true*) for 000 and for 001.

Intersection and union of sets of assignments are translated simply into conjunction and disjunction of their BDD representation, respectively; complementation becomes BDD negation. We will denote the Boolean BDD operators as **and**, **or** and **not**. To implement the existential operators $\exists x$, we use the BDD existential operators over the Boolean variables $x_1 \ldots x_n$ that represent (the enumerations of) the values of x. Thus, if B_η is the BDD representing the assignments satisfying the subformula η in the current state of the monitor, then $\textbf{exists}(x, B_\eta) = \exists x_1 \ldots \exists x_k B_\eta$ is the BDD that represents the assignments satisfying $\exists x \eta$. Finally, $\mathrm{BDD}(\bot)$ and $\mathrm{BDD}(\top)$ are the BDDs that return always 0 or 1, respectively.

The RV algorithm for a QTL formula φ based on BDDs is as follows:

1. Initially, for each $\eta \in sub(\varphi)$ of the specification φ, $\mathsf{now}(\eta) = \mathrm{BDD}(\bot)$.
2. Observe a new event $p(a)$ as input; $hash(a)$
3. Let $\mathsf{pre} := \mathsf{now}$.
4. Make the following updates for the formulas $sub(\varphi)$, where if $\psi \in sub(\eta)$ then $\mathsf{now}(\psi)$ is updated before $\mathsf{now}(\eta)$.
    - $\mathsf{now}(true) = \mathrm{BDD}(\top)$
    - $\mathsf{now}(p(a)) =$ if current event is $p(a)$ then $\mathrm{BDD}(\top)$ else $\mathrm{BDD}(\bot)$
    - $\mathsf{now}(p(x)) =$ if current event is $p(a)$ then $\textbf{build}(x, val(a))$ else $\mathrm{BDD}(\bot)$
    - $\mathsf{now}((\eta \wedge \psi)) = \textbf{and}(\mathsf{now}(\eta), \mathsf{now}(\psi))$
    - $\mathsf{now}(\neg\eta) = \textbf{not}(\mathsf{now}(\eta))$
    - $\mathsf{now}((\eta \,\mathcal{S}\, \psi)) = \textbf{or}(\mathsf{now}(\psi), \textbf{and}(\mathsf{now}(\eta), \mathsf{pre}((\eta \,\mathcal{S}\, \psi))))$
    - $\mathsf{now}(\ominus \eta) = \mathsf{pre}(\eta)$
    - $\mathsf{now}(\exists x\, \eta) = \textbf{exists}(\langle x_0, \ldots, x_{k-1} \rangle, \mathsf{now}(\eta))$
5. Goto step 2.

For a subformula η of the specification, $\mathsf{now}(\eta)$ is the BDD representation of $I[\eta, \sigma, i]$ according to the set semantics. The output of the algorithm after a given trace corresponds to the value of $\mathsf{now}(\varphi)$. Accordingly, it will be *true* if this value is $\mathrm{BDD}(\top)$ and *false* if it is $\mathrm{BDD}(\bot)$.

2.3 Predictive Runtime Verification

While monitoring an execution of a system against a formal specification, it is sometimes beneficial to be able to predict forthcoming possible results. The RV algorithm for QTL provides a *true*/*false* output for each prefix that is observed. The output can alternate between these truth values for subsequent prefixes. It is sometimes useful to be able to predict the possible outputs for extensions of the current trace, e.g., a possible future *false* output that corresponds to some potential problem. Then, one may apply some measures to alleviate such a situation, either by imposing some control on the system or by performing an abort.

Classical definitions for RV over temporal properties, e.g., [7,21], suggest calculating a conclusive *verdict* of *success* or *failure*, respectively, when *all* the extensions of the current trace into a full execution agree w.r.t. *satisfying* or *not satisfying*, respectively, the property. In particular, this can be useful if such a verdict can be decided based on a minimal prefix. For past temporal logic, a *true*/*false* output is given based on the currently monitored prefix. The outputs can alternate over subsequent prefixes. *Aniticipatory RV* [20] generalizes this, and looks at the possible outputs after a given prefix and the (minimal and maximal) distances to them (including the distance ∞) and provides an algorithm for the propositional version of the logic. We are interested here in calculating the possible outputs for all extensions of the current trace, limited to k additional events, where k is fixed, for the *first-order* past LTL QTL. We will show in Sect. 4 that making a prediction about *all* the extensions of a QTL property, without a given bound, is undecidable.

A naive k-step prediction algorithm would check, after the currently inspected trace, the possible extensions of up to k events. For each such extension, the RV algorithm is applied, continuing from the current prefix, to provide a verdict. Depending on the interpretation, a subsequent *false* output can mean a *failure* verdict, which can be sufficient to stop the generation of longer or further extensions and take some preventing action. Obviously, this method is impractical: even if the number of possible events extending a single trace by one step is finite, say n, its time complexity is $O(n^k)$. For the propositional case, this may be feasible when the specification involves only a few propositions (with m propositions, one can form $n = 2^m$ events). However, for the case of events with data, n can be enormous, or even unbounded.

2.4 Isomorphism over Relations Representing QTL Subformulas

The main challenge in predicting the future outputs for a trace is to restrict the number of cases one needs to consider when extending it. It can be argued that one can limit the number of possible events extending a trace by a single step to the values that appeared so far; in addition, for values that *did not* appear so far in the trace, a single representative is enough (but after using that representative in the current event, one needs a fresh representative for the values not seen so far, and so forth). However, in this case, the number of relevant events increases as the trace increases (although some clever use of garbage collection [17] may sometimes decrease the relevant values), which can result in a large number of values after a long trace.

Our proposed prediction method is based on calculating equivalence relations on the observed data values that guarantee the following: an extension of the currently observed trace can be simulated by an extension of the same length and with the same verdict when replacing in the next observed event an occurring data value with an equivalent one.

Let $R \subseteq \mathcal{D} = D_1 \times \ldots \times D_n$ be a relation over multiple (not necessarily distinct) domains obtained as the set semantics. Recall from sets semantics that if $R = I[\eta, \sigma, i]$ then it represents the assignments that satisfy the specification η at the i^{th} position of the trace σ. In this context, each tuple in R is an assignment for the free variables of η. Thus, each component D_i is associated with some variable $x \in \mathit{free}(\eta)$. Let $f^x : D_i \mapsto D_i$ be a function over D_i, where the i^{th} component of the relation R is associated with the variable x. We abuse notation and denote by $f^x(\tau)$ also the extension of f^x to a tuple $\tau \in \mathcal{D}$, which changes only the D_i component in the tuple according to f^x. Furthermore, denote by $f^x[R]$ the application of f^x to each tuple in R. We say that R and R' are *isomorphic* with respect to f^x if $R' = f^x[R]$ for an injective and surjective function f^x. If $R = f^x[R]$, then we say that R is an *automorphism* with respect to f^x.

Lemma 1. *If R_i and $R_i{}'$ are isomorphic (automorphic) w.r.t. f^x for $i \in \{1,2\}$, then also the following are isomorphic with respect to f^x:*

- *$\overline{R_i}$ (the complement of R_i),*
- *$R_1 \cap R_2$ and $R_1{}' \cap R_2{}'$ and*
- *$R_1 \cup R_2$ and $R_1{}' \cup R_2{}'$.*

Denote by $f^x_{a\leftrightarrow b}$ the function that replaces a with b and vice versa, and does not change the other values. Denote by $R_{x=a}$ the restriction of the relation R to tuples where their x component has the value a. The following lemma provides a condition for deciding automorphism.

Lemma 2. *$f^x_{a\leftrightarrow b}$ is an automorphism over R if $R_{x=a} = R_{x=b}$.*

Denote by $E[\eta,\sigma,x]$ the equivalence relation w.r.t. the variable $x \in \mathit{free}(\eta)$ for a subformula η of a given specification φ, constructed from $R = I[\eta,\sigma,i]$ where $i = |\sigma|$. That is,

$$E[\eta,\sigma,x] = \{(a,b) \mid R_{x=a} = R_{x=b}\}. \qquad (1)$$

Let t be some type of variables allowed in the specification. Then, let

$$\mathcal{E}[\sigma,t] = \bigcap_{x \in \mathit{free}(\eta) \wedge \mathit{type}(x)=t \wedge \eta \in \mathit{sub}(\varphi)} E[\eta,\sigma,x]. \qquad (2)$$

We need to take care of the following special case. Let $r(a)$ appears in the specification for some constant a. Then a can only be equivalent to itself, since $I[r(a),\ \sigma.r(a),\ i] \neq I[r(a),\ \sigma.r(b),\ i]$ for any $b \neq a$ for events $r(a)$ and $r(b)$. (A similar argument holds for an event with more arguments, e.g., $r(y,a,b)$.) For simplicity, the following descriptions will refer to events with a single argument (as we did in the definition of the syntax of QTL).

Lemma 3. *Let $(a,b) \in \mathcal{E}[\sigma,t]$ and r is a predicate over a parameter of type t. Then, $f^x_{a\leftrightarrow b}$ is an isomorphism between the relations in the summary after $\sigma.\{r(a)\}$ and $\sigma.\{r(b)\}$.*

Proof. By construction, $f^x_{a\leftrightarrow b}[R]$ is an automorphism for each relation $R = I[\sigma,\eta,i]$ in the summary. The result is obtained using Lemma 1 by induction on the number of operators that need to be applied to construct the relations $I[\sigma.r(a),\eta,i+1]$ and $I[\sigma.r(b),\eta,i+1]$ in the subsequent summary, according to the set semantics, from the relations calculated for σ. Note that because of using different singleton relation for the event $r(a)$ extending the current trace and a different singleton relation for the event $r(b)$, the *automorphism* calculated from the original summary results is an *isomorphism* for the subsequently constructed relations rather than an automorphism. □

Lemma 4. *Let $(a,b) \in \mathcal{E}[\sigma,t]$. Then for each finite trace $\sigma.r(a).\rho$ there exists a trace $\sigma.r(b).\rho'$ such that these traces result in the same verdict, and ρ and ρ' have the same length.*

Proof. We construct the extension ρ' as follows: for each term that appears in an event of ρ, we construct a corresponding term with the same predicate, and with a replaced with b and vice versa, and other values unchanged. Then the result is obtained by induction on the length of the considered extensions. The induction step (including the first step after σ) is obtained using Lemma 3. □

Consequently, it is redundant to generate two extensions $r(a)$ and $r(b)$ for a trace σ where $(a,b) \in \mathcal{E}[\sigma,t]$. Applying Lemma 4 repeatedly from σ results in the following recursive procedure for predicting RV. From every extension of σ up to k events, it runs on every predicate r and extends it with a single value for every equivalence class from $\mathcal{E}[\sigma,t]$. The principle algorithm then appears in Algorithm 1. In this version, the algorithm stops and produces a *failure* verdict whenever one of the extensions of the current trace *falsifies* the specification φ. Other variants can exit upon *satisfying* φ, or continue to check whether both *true* and *false* are attainable.

Algorithm 1. Pseudocode for the prediction algorithm

```
procedure PREDICT($\sigma, k$)
    for each type $t$ in the specification do
        $E \leftarrow \mathcal{E}[\sigma,t]$
        while $E \neq \emptyset$ do
            let $[a] \in E$                                   ▷ $[a]$ is the eq. class containing $a$.
            for each predicate $r$ over parameter of type $t$ do
                generate an event $r(a)$
                apply RV to update summary from $\sigma$ to $\sigma.r(a)$
                if RV output is false then
                    exit("failure verdict")
                if $k > 1$ then
                    PREDICT($\sigma.r(a), k-1$)
            $E \leftarrow E \setminus [a]$
```

An (implemented) extension of the described algorithm allows the predicates to have multiple parameters as follows. Equivalence classes are calculated independently for each parameter, and each event includes a representative for each corresponding equivalence class. Hence, in $write(f, d)$, we select a representative for f and a representative for d, making the number of cases the product of the two equivalence classes used.

The calculation of $\mathcal{E}[\sigma,t]$ involves intersecting equivalence classes of relations associated with the assignments for the free variables that satisfy subformulas of φ. Each individual equivalence class is calculated with respect to a free variable with the type t. Refining the type definitions in the formula can result in checking less representatives. Consider for example the following property $\forall f\ ((\exists d\ write(f,d)) \rightarrow$

$(\neg close(f)\ S\ open(f)))$. Both variables f and d can be originally defined with type strings. However, the f operator corresponds to a *file-name*, and the d operator corresponds to *data*. If we duplicate the type *string* into these two copies, we can achieve a more efficient prediction, where representatives for the file names observed would be used solely for f and representatives for the data values observed would be used for d.

Duplicating of types associated with variables, and corresponding also to constants and parameters of predicates, can be automated. First, rename the variables so that each quantified variable appears exactly once (this does not change the meaning of the formula). Now, observe the following principle: all the variables that appear within the same predicate (and in the same position, if the predicate has multiple parameters), must have the same type. Now, if the same variable appears (in the same position) in different predicates and within the same scope of quantification, then forcing variables to have the same type based on the above principle can diffuse to other occurrences of these predicates.

A graph algorithm can then be used for automating type duplication. The nodes of the graph are labeled by either variables or predicates (for predicates with multiple parameters, such a node will include the predicate *and* the position of the relevant parameter, respectively). Undirected edges will connect variables with the predicates (with positions, respectively) in which they occur. Then, all the variables in a maximal connected subgraph must have the same type, whereas a type that is shared by multiple such subgraphs can be duplicated in order to refine the calculation of the equivalences. Consider the following formula $(\exists x \ominus (q(x) \vee r(x)) \wedge (\forall y \exists z(\ominus r(y) \rightarrow \ominus q(z)) \vee \exists u \ominus p(u)))$. Then according to the constructed graph, which appears in Fig. 1, the variables x, y and z need to be of the same type, but u can have a different type.

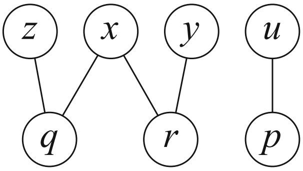

Fig. 1. Variables/Predicates dependency graph

An Example of Equivalence Class Partitioning. Consider the following property $\exists x\,(\ominus\ q(x) \wedge \neg \ominus\ r(x))$ and the trace $q(1).q(2).r(1).q(3).q(4).q(5).r(2).r(3).r(4).q(6)$

Table 1 presents relations that correspond to the subformulas in the summary. The table shows these relations in now after a trace that includes the first event, the first two events, all but the last event, and the entire trace. The letter U represents the set of all possible values for x (this can be, e.g., the natural numbers). The last column presents the equivalence relations, as defined in Eq. (1), calculated per each relation at the end of the trace.

The intersection of the equivalence relations that appear in the last column of the table gives the following equivalence classes:

$$\{\{1,2,3,4\},\{5\},\{6\},U\setminus\{1,2,3,4,5,6\}\}.$$

Table 1. Calculating relations and equivalence classes for the example property

Subformula	Event					
	$q(1)$	$q(2)$	$\dots$	$r(4)$	$q(6)$	Eq. classes
$q(x)$	$\{1\}$	$\{2\}$	$\dots$	$\emptyset$	$\{6\}$	$\{\{6\}, U \setminus \{6\}\}$
$\diamondminus\, q(x)$	$\{1\}$	$\{1,2\}$		$\{1,2,3,4,5\}$	$\{1,2,3,4,5,6\}$	$\{\{1,2,3,4,5,6\}, U \setminus \{1,2,3,4,5,6\}\}$
$r(x)$	$\emptyset$	$\emptyset$		$\{4\}$	$\emptyset$	$\{U\}$
$\diamondminus\, r(x)$	$\emptyset$	$\emptyset$		$\{1,2,3,4\}$	$\{1,2,3,4\}$	$\{\{1,2,3,4\}, U \setminus \{1,2,3,4\}\}$
$\neg \diamondminus\, r(x)$	U	U		$U \setminus \{1,2,3,4\}$	$U \setminus \{1,2,3,4\}$	$\{\{1,2,3,4\}, U \setminus \{1,2,3,4\}\}$
$\diamondminus\, q(x) \wedge \neg \diamondminus\, r(x)$	$\{1\}$	$\{1,2\}$		$\{5\}$	$\{5,6\}$	$\{\{5,6\}, U \setminus \{5,6\}\}$

This is the equivalence relation defined in Eq. (2) that is used to select the representatives for extending the trace.

3 Prediction Using BDD Representation

We saw in the previous section how to define equivalence classes on data values that would lead to extensions of the current trace with the same lengths and verdicts. We established that it is sufficient to select a single representative from each equivalence class. Calculating these equivalence classes explicitly can be complex and time consuming. Instead, we show how to take advantage of the BDD representation to calculate the equivalence classes. We encode these equivalence classes using Boolean formulas that involve the components of the summary, calculated during the RV algorithm, which are also BDDs. These formulas can be used to calculate the equivalence classes using a BDD package. Then, selecting a representative from the equivalence class and updating the remaining equivalence classes are also implemented using operators on BDDs.

Recall that the RV algorithm described in Sect. 2.2 calculates BDDs that correspond to the assignments for the subformulas after an inspected trace σ of length i. That is, for a subformula η, the BDD $\mathsf{now}(\eta)$ represents the relation $R = I[\eta, \sigma, i]$ containing the assignments to the free variables of R that satisfy η at position $i = |\sigma|$ of the trace σ. Suppose that R is such a BDD B in the summary, with the bits $x_1, \dots, x_n$ representing (enumeration) values of the D_i component of R, and with the bits $y_1, \dots, y_m$ representing the rest of the components.

Implementing the algorithm using BDDs, we start by translating the condition of Lemma 2 for automorphism into a check that can be automated using BDDs. Let B be the BDD representation of $R = I[\eta, \sigma, i]$, let $a_1, \dots, a_n$ be the bit vector that represents the value (enumeration) a, and let $b_1, \dots, b_n$ be the bit vector that represents the value b. Then $f^x_{a \leftrightarrow b}$ is an automorphism over R iff the following BDD formula evaluates to *true*[2]:

$$\forall y_1 \dots \forall y_m ((B[x_1 \setminus a_1] \dots [x_n \setminus a_n]) \leftrightarrow (B[x_1 \setminus b_1] \dots [x_n \setminus b_n])) \tag{3}$$

[2] $(\eta \leftrightarrow \psi)$ is a shorthand for $((\eta \rightarrow \psi) \wedge (\eta \leftarrow \psi))$. Also $B[c \setminus d]$ denotes the BDD where the value of the bit c in the BDD is set constantly to d.

Next, we generate from a BDD B, representing some relation $R = I[\eta, \sigma, i]$ in the summary, a representation of the equivalence classes of $E[\eta, \sigma, x]$. This is an implementation of Equation (1) using BDDs. The bit vectors $g_1, \ldots, g_n$ and $h_1, \ldots, h_n$ represent pairs of values g and h for the variable x such that $f_{g \leftrightarrow h}$ is an isomorphism for R. That is $R_{x=g} = R_{x=h}$ for each pair of values g and h. We denote this formula by $GH[B,x]$.

$$\begin{aligned} GH[B,x] = \forall y_1 \ldots \forall y_m (\exists x_1 \ldots \exists x_n (B \wedge (x_1 \leftrightarrow g_1) \ldots \wedge (x_n \leftrightarrow g_n)) \leftrightarrow \\ \exists x_1 \ldots \exists x_n (B \wedge (x_1 \leftrightarrow h_1) \ldots \wedge (x_n \leftrightarrow h_n))) \end{aligned} \tag{4}$$

An alternative and more efficient method for obtaining the BDD $GH[B,x]$ is to employ the simplified formula, which avoids using the existential quantification.

$$GH[B,x] = \forall y_1 \ldots \forall y_m (B[x_1 \setminus g_1] \ldots [x_n \setminus g_n] \leftrightarrow B[x_1 \setminus h_1] \ldots B[x_n \setminus h_n]) \tag{5}$$

Now we can construct a BDD representation for the equivalence classes $\mathcal{E}[\sigma, t]$, as defined in Eq. (2). We need to take, for each type t the conjunction of all $GH[B_\eta, x]$, for $\eta \in sub(\varphi)$, $x \in free(\varphi)$ and $type(x) = t$, and where $B_\eta = \mathsf{now}(\eta)$ in the summary of the RV algorithm.

$$GH = \bigwedge_{x \in free(\eta) \wedge type(x)=t \wedge \eta \in sub(\varphi)} GH[B_\eta, x] \tag{6}$$

4 Undecidability of Unbounded Prediction

We presented an algorithm for calculating the verdicts that can be obtained by extending the inspected trace, checked against a first-order past LTL specification, by up to k steps. In this section we will show that making such a prediction *without a length restriction* is undecidable[3].

The proof is by reduction from the undecidable *post correspondence problem* (PCP). An instance of the PCP problem consists of two *indexed* sets T_1 and T_2, each of $n > 0$ nonempty *words* from over some finite alphabet Σ. The problem is to decide whether there is a non-empty finite sequence $i_1, i_2, \ldots, i_k$ of indexes, where each $i_j \in [1..|T_1|]$, such that $T_1(i_1).T_1(i_2) \ldots T_1(i_k) = T_2(i_1).T_2(i_2) \ldots T_2(i_k)$ (using the concatenation operator "."). That is, whether concatenating words from T_1 and from T_2, with possible repeats , according to some common sequence of indexes, gives the same string.

For example, consider an instance of PCP where $T_1 = \{(aa, 1), (abb, 2), (aba, 3)\}$ and $T_2 = \{(baa, 1), (aab, 2), (ab, 3)\}$, where each pair includes a word and its corresponding index. Thus, we can write, e.g., $T_1(2) = abb$. For this instance of PCP, there is a simple solution, where each word appears exactly once; when concatenating the words with index order 3 2 1, we obtain $T_1(3).T_1(2).T_1(1) = aba.abb.aa$, and for $T_2(3).T_2(2).T_2(1) = ab.aab.baa$, resulting in the same string

$$abaabbaa. \tag{7}$$

[3] A proof of undecidability of first-order *future* LTL that includes *interpreted* and *uninterpreted relation* and *function* symbols is shown in [6]. Note that our logic is far more restrictive than that.

The reduction constructs from each instance of the PCP problem a past first-order LTL formula that is satisfiable by a trace if and only if it describes a solution to the problem. The trace simulates a concatenating of words from T_1 and T_2. The input includes, except for the sequence of letters, additional information that allows breaking the string according to the tokens of T_1 and according to the tokens of T_2.

Adjacent to each letter in the string, we add two values, from some unbounded events, which delimit the individual strings for T_1 and for T_2, correspondingly. For example, given the delimiting values p, q, r, we obtain from the above concatenated word in (7) the following sequence of triples, each consisting of a letter from Σ, and two delimiters, for T_1 and T_2:

$$\overbrace{\underbrace{\rlap{$\overbrace{(a,p,p)(b,p,p)(a,p,q)}^{\text{p}}\overbrace{(a,q,q)(b,q,q)(b,q,r)}^{\text{q}}\overbrace{(a,r,r)(a,r,r)}^{\text{r}}$}\underbrace{}_{\text{p}}\underbrace{}_{\text{q}}\underbrace{}_{\text{r}}}_{T_2}}^{T_1}. \tag{8}$$

In each triple, the first component is the letter, the second is the delimiting value for T_1 and the third component is the delimiting value for T_2.

The delimiting value will appear for both strings in T_1 and of T_2 *in the same order*, to impose the restriction that the same sequence of indexes are used. Thus, in (8), p appears before q and q appears before r. A delimiting value will not repeat after having followed the letters for a single appearance of a string from T_1, and similarly for T_2, even if the same string repeats in the concatenation. The temporal specification will enforce that delimiting of a word from T_1 and delimiting of a word from T_2 with the same value (e.g., the value q) to words with the same indexes in T_1 and T_2 respectively. In the above example, the delimiting value q corresponds to words of T_1 and T_2 with the index 2.

Now, to represent such as sequence as an input trace for RV, each of the above triples will correspond to a successive triple of events, each of the form $l(x).t_1(v_1).t_2(v_2)$. Finally, the trace ends with a single parameterless event e. The predicates in the monitored events have the following roles:

e with no parameters (a proposition). It designates the end of the sequence representing a solution for the PCP problem.

$l(x)$ This is a letter within the concatenation. Since the two concatenations of words need to produce the *same* string, there is only a single $l(x)$ event in each triple.

$t_1(v_1)$ v_1 is a delimiting value for the currently observed word from T_1. Similarly,

$t_2(v_2)$ v_2 is a delimiting value for the currently observed word from T_2.

Then, the sequence (8) become the following sequence of events:

$$l(a).t_1(p).t_2(p).\ \ l(b).t_1(p).t_2(p).\ \ l(a).t_1(p).t_2(q).\ \ l(a).t_1(q).t_2(q).$$
$$l(b).t_1(q).t_2(q).\ \ l(b).t_1(q).t_2(r).\ \ l(a).t_1(r).t_2(r).\ \ l(a).t_1(r).t_2(r).\ \ e.$$

Then, we construct a QTL formula φ as the concatenation of the following conditions:

- Value v_1 (v_2, respectively) within the predicates t_1 (t_2, respectively) can only appear in adjacent triples. Once this value is replaced by a different value in the next triple, the old value never returns.
- The order between the v_1 values and the v_2 values is the same, that is, $t_1(p)$ appears before $t_1(q)$ if and only if $t_2(p)$ appears before $t_2(q)$.
- Each concatenation of letters $l(x)$ from subsequent triples, that is limited by events of the form $t_1(p)$ for some value p forms a word $T_1(i)$ for some i. Similarly, the concatenation of letters $l(x)$ limited by $t_2(q)$ forms a word $T_2(j)$ for some j. Furthermore, if $p = q$ then $i = j$.

Now, φ is satisfied by a trace σ if σ describes a solution for the PCP instance. Hence, predicting when there is a *true* outcome for an extension of the empty trace is undecidable. The undecidability proof suggests that our algorithm for predicting the verdict in k steps gives a compromise for this kind of long-term prediction.

Our undecidability proof has several additional consequences. Temporal safety [22] properties can be written as $\Box\psi$ (see [24]), where $\Box$ stands for the standard LTL operator *always*, i.e., *for each prefix*, and ψ contains only past modalities. It follows from the above construction that the satisfiability of the first-order temporal safety properties of the form $\Box\psi$ for a past ψ is undecidable: just take $\psi = \neg\varphi$ with the above constructed formula φ, which is satisfiable exactly when the instance of the PCP problem does not have a solution. For propositional LTL, it is useful to conclude a *success* (*failure*, respectively) verdict based on monitoring a *minimal* prefix of a trace, when all of its infinite extensions satisfy (does not satisfy, respectively) the property. Such an algorithm appears in [21]. It follows from our construction that this is undecidable for the first-order case. This also gives some explanation of why RV tools for first-order past LTL, such as DejaVu and MonPoly provide a *true*/*false* outputs for past properties ψ, instead of checking $\Box\psi$.

5 Experiments

In order to assess the effectiveness and efficiency of our algorithm, which we term iPRV (isomorphic Predictive RV), we extended DejaVu to incorporate our prediction approach. The experiments were performed on an Apple MacBook Pro laptop with an M1 Core processor, 16 GB RAM, and 512 GB SSD storage, running the macOS Monterey operating system. We carried out a comparative analysis against the straightforward brute-force prediction method, which was also integrated into DejaVu. We expressed properties, four of them are shown in Fig. 2, using DejaVu's syntax, and evaluated the tool's performance based on time, and the number of prediction extensions (we termed *cases* or C) used. We repeated experiments with traces of diverse sizes and events order. To measure performance and the influence of the size of parameter k, we experimented with different sizes of k. We used two different approaches when conducting experiments: some experiments stopped once the expected verdict *false* was reached, while others ran until all possible extensions, either exhaustively, for brute-force, or based on representatives with our algorithm, were examined (unless a time limit was surpassed). The unstopped experiments simulated the worst-case scenario when the expected verdict was not found. All the experiments in this section, along with their specifications and the corresponding traces, including further examples, are available in our GitHub repository [30].

```
P1.  exists x . (@p(x) & g(x) &
        H (p(x) → (!@p(x) & !(@(@(p(x )))))) &
        H (g(x) → (!@g(x))))

P2.  forall x . (r(x) →
        exists y . (!q(y) S p(y)))

P3.  forall f . ((exists d . write(f,d)) →
        (!close(f) S open(f)))

P4.  exists x . (P q(x) & !P r(x))
```

1. $\varphi_1 = \exists x\,(\ominus p(x) \wedge g(x) \wedge \boxminus\,(p(x) \rightarrow (\neg\ominus p(x) \wedge \neg(\ominus(\ominus(p(x))))) \wedge \boxminus\,(g(x) \rightarrow (\neg\ominus g(x))))$
2. $\varphi_2 = \forall x\,(\,r(x) \rightarrow \exists y(\neg q(y)\; S\; p(y)))$
3. $\varphi_3 = \forall f\;((\exists d\; write(f,d)) \rightarrow (\neg close(f)\; S\; open(f)))$
4. $\varphi_4 = \exists x\,(\diamondminus\, q(x) \wedge \neg \diamondminus\, r(x))$

Fig. 2. Evaluated properties in DejaVu (left) and QTL (right) formalism. (The QTL operators $\boxminus$ and $\diamondminus$ are denoted in DejaVu as **H** and **P** respectively)

Traces. Distinct trace files, τ_{min}, τ_{med}, and τ_{max}, were generated for each property, to evaluate iPRV approach. Those files contain random traces in which the order of the events along with their values were set in randomly. Moreover, the *min*, *med*, and *max* descriptors associated with each file represent the size of the trace. The diversity of traces enabled us to perform a thorough analysis and comparison of the performance of iPRV in comparison to the trivial brute-force approach, under different trace sizes and events order for each property.

The τ_{min} traces consist of small traces with less than 15 events each; they provide small equivalence classes. The τ_{med} traces consist of up to 150 events, while the τ_{max} trace can contain up to 1000 events.

For example, for property P4, the traces were created as a random sequence of events with predicates q and r, with data that was randomly generated within a specified range. However, a constraint was applied such that the number of all the q events does not exceed in every prefix of the generated trace the number of r events by more than 5. This guarantees the violation of property P4 for $k \geq 6$.

Results. Tables 2 and 3 summarize part of our experiment results. They illustrate the efficacy of the iPRV, our proposed prediction algorithm, in comparison to the straightforward brute-force approach. The use of the ∞ symbol indicates instances where the prediction process exceeded 1000 seconds. Additionally, C denotes the number of extensions calculated during a single prediction.

Table 2 displays the results of the experiments where the prediction process was executed until all possible outcomes were found, which simulates the worst-case scenario when the expected verdict is not found. Table 3 offers a comparative on-the-fly analysis, where the prediction process stops upon the discovery of a failure. Not surprisingly, as the prediction horizon increases from $k = 4$ to $k = 5$, both methods take more time to complete the executions. The brute-force method, in particular, struggles to complete the executions as complexity rises. From the results, we can conclude that iPRV is at least a few times faster than the brute-force method.

In few cases, in particular where the trace is short and the prediction horizon is not large, the speed improvement was not very significant, but still iPRV is faster; whereas in some comparisons it takes almost the same time. In other cases, it is four times faster and even much more. For example, in Table 2, for property P1, the iPRV method with $k = 4$ and τ_{med} took 0.05 seconds, while in the same configuration, but for the brute-force method, it took more than 131.54 seconds, which means that iPRV is approximately 2600 times faster in this case. In other cases, the brute-force method did not stop within the time frame of 1000 seconds while iPRV managed to stop with a calculation time that is significantly shorter than the 1000 seconds. In these cases, the speed improvement is several orders of magnitude (assuming, optimistically, a 1000 seconds execution time for the brute-force executions). Furthermore, the number of extensions (denoted in the table by C) required by the iPRV method to provide the prediction is significantly less than for the brute-force method. The gap between these methods intensifies as the prediction horizon k increases.

Table 2. Comparison where both methods run fully

Property	Method	Trace τ_{min}	Trace τ_{med}	Trace τ_{max}
$P1$	iPRV ($k = 4$)	0.06 s, 1,280 C	0.05 s, 1,168 C	0.10 s, 2,184 C
	Brute ($k = 4$)	0.07 s, 2,416 C	131.54 s, 108,496,240 C	∞
	iPRV ($k = 5$)	0.21 s, 7,816 C	0.15 s, 6,984 C	0.31 s, 13,304 C
	Brute ($k = 5$)	0.24 s, 21,568 C	∞	∞
$P2$	iPRV ($k = 4$)	0.09 s, 2,240 C	0.07 s, 1,856 C	0.12 s, 2,240 C
	Brute ($k = 4$)	0.81 s, 296,631 C	∞	∞
	iPRV ($k = 5$)	0.23 s, 15,292 C	0.21 s, 12,672 C	0.29 s, 15,296 C
	Brute ($k = 5$)	6.62 s, 7,103,366 C	∞	∞
$P3$	iPRV ($k = 4$)	0.04 s, 864 C	0.03 s, 576 C	0.04 s, 864 C
	Brute ($k = 4$)	0.05 s, 1,504 C	0.64 s, 259,840 C	∞
	iPRV ($k = 5$)	0.14 s, 5,184 C	0.10 s, 4,352 C	0.17 s, 5,184 C
	Brute ($k = 5$)	0.15 s, 10,736 C	1.21 s, 2,538,680 C	∞
$P4$	iPRV ($k = 4$)	0.12 s, 4,388 C	0.14 s, 4,388 C	0.19 s, 4,388 C
	Brute ($k = 4$)	0.50 s, 170,304 C	∞	∞
	iPRV ($k = 5$)	0.40 s, 37,512 C	0.45 s, 37,512 C	0.69 s, 37,512 C
	Brute ($k = 5$)	3.29 s, 3,558,752 C	∞	∞

Table 3. Comparison where both methods stopped at the expected verdict

Prop	Method	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
$P1$	iPRV	0.02 s, 1 C	0.01 s, 1 C	0.03 s, 1 C	0.03 s, 1 C	0.04 s, 1 C
	Brute	0.05 s, 1 C	0.09 s, 1 C	0.14 s, 1 C	0.27 s, 1 C	0.39 s, 1 C
$P2$	iPRV	0.29 s, 15,296 C	∞	∞	∞	∞
	Brute	∞	∞	∞	∞	∞
$P3$	iPRV	0.02 s, 3 C	0.03 s, 3 C	0.08 s, 3 C	0.05 s, 3 C	0.08 s, 3 C
	Brute	1.31 s, 819 C	1.51 s, 819 C	2.29 s, 819 C	5.19 s, 819 C	10.22 s, 819 C
$P4$	iPRV	0.62 s, 37,512 C	4.24 s, 446,857 C	6.10 s, 668,869 C	6.87 s, 668,869 C	6.58 s, 668,869 C
	Brute	∞	∞	∞	∞	∞

6 Conclusion

In this work, we presented an algorithm for predicting the possible future outputs during RV of executions with data; the specification is written in past time first-order linear temporal logic QTL and the prediction is limited to the next (parametric) k events. This can be used for preventive actions to be taken before an unrecoverable situation occurs. For efficiency, our algorithm calculates equivalence classes of values that produce isomorphic extensions to the currently observed trace. This allows exploring only representative extensions in order to perform predictions, avoiding the inefficient naive method that checks all the possible event sequences for the next k steps. We demonstrated how to leverage from the BDDs representations for efficient construction and representation of these equivalence classes. The algorithm was implemented as an extension of the RV tool DejaVu.

We have shown that, unlike propositional temporal logic, prediction for past first-order temporal specification *without* a fixed length limit is an undecidable problem. This was proved using a reduction from the post correspondence problem (PCP). This makes the k-step prediction a decidable compromise.

The experimental results indicate that our proposed algorithm significantly outperforms the brute-force method in terms of time and the number of prediction cases calculated during the prediction process; in certain cases, our prediction method successfully concluded its prediction, while the brute-force approach persisted in running without attaining completion within a reasonable time frame.

Although our experimental results show that the speed of our algorithm is far better than the brute-force approach, prediction can still be a significant time consuming task. Whereas the incremental processing after each event in DejaVu takes typically microseconds [15], the incremental complexity for prediction for, e.g., $k = 4$ can take a significant fraction of a second, and for a larger prediction horizon it can further grow. This of course depends on the property and the observed trace. Thus, a naive use of the prediction algorithm with a not so large prediction horizon should be able, in principle, to online monitor traces with the arrival speed of a few events per second. We propose that prediction should be used in combination with other methods that allow restricting the future executions of the observed system, e.g., when an approximated model of the system is available or is obtained using learning techniques (see, e.g., [27,28]). Moreover, prediction should be delegated to a concurrent task, so that normal (prediction-less) monitoring would not be delayed, in case of a quick burst of newly observed events.

Acknowledgements. The authors would like to thank Panagiotis Katsaros for insightful comments on a preliminary draft of the paper.

References

1. Allan, C., et al.: Adding trace matching with free variables to AspectJ. In: OOPSLA 2005, SIGPLAN Notices, vol. 40, no. 10, pp. 345–364. ACM (2005)
2. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_5
3. Barringer, H., Havelund, K.: TRACECONTRACT: a scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_7
4. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from EAGLE to RULER. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 111–125. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77395-5_10
5. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 45 (2015)
6. Bauer, A., Küster, J.-C., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 59–75. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_4
7. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 126–138. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77395-5_11
8. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
9. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS, pp. 428–439 (1990)
10. Cimatti, A., Tian, C., Tonetta, S.: Assumption-based runtime verification of infinite-state systems. In: Feng, L., Fisman, D. (eds.) RV 2021. LNCS, vol. 12974, pp. 207–227. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88494-9_11
11. Colombo, C., Pace, G.J., Schneider, G.: LARVA - safer monitoring of real-time java programs (tool paper). In: SEFM 2009, pp. 33–37. IEEE Computer Society (2009)
12. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. J. Softw. Tools Technol. Transf. **18**(2), 205–225 (2016)
13. D'Angelo, B., et al: LOLA: runtime monitoring of synchronous systems. In: TIME 2005, pp. 166–174 (2005)
14. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. **5**(2), 192–206 (2012)
15. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. In: FMCAD 2017, pp. 116–123 (2017)
16. Goubault-Larrecq, J., Olivain, J.: A smell of ORCHIDS. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 1–20. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89247-2_1

17. Havelund, K., Peled, D.: BDDs on the run. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 58–69. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_8
18. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
19. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 61–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_3
20. Kallwies, H., Leucker, M., Sánchez, C., Scheffel, T.: Anticipatory recurrent monitoring with uncertainty and assumptions. In: Dang, T., Stolz, V. (eds.) RV 2022. LNCS, vol. 13498, pp. 181–199. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17196-3_10
21. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods Syst. Design **19**(3), 291–314 (2001)
22. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3**(2), 125–143 (1977)
23. McMillan, K.L.: Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic Publishers (1993)
24. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems - Specification, pp. I–XIV, 1–427. Springer, New York (1992). https://doi.org/10.1007/978-1-4612-0931-7. ISBN 978-3-540-97664-6
25. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. J. Softw. Tools Technol. Transf. **14**, 249–289 (2011). https://doi.org/10.1007/s10009-011-0198-6
26. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 596–610. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_55
27. Yu, K., Chen, Z., Dong, W.: A predictive runtime verification framework for cyber-physical systems. In: SERE (Companion), pp. 223–227 (2014)
28. Zhang, X., Leucker, M., Dong, W.: Runtime verification with predictive semantics. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 418–432. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_37
29. JavaBDD. https://javabdd.sourceforge.net
30. iPRV DejaVu tool source code. https://github.com/moraneus/iPRV-DejaVu
31. DejaVu tool source code. https://github.com/havelund/dejavu