TP-DejaVu: Combining Operational and Declarative Runtime Verification*

Klaus Havelund[1], Panagiotis Katsaros[2], Moran Omer[3], Doron Peled[3], and Anastasios Temperekidis[2]

[1] Jet Propulsion Laboratory, California Inst. of Technology, Pasadena, USA
[2] Aristotle University of Thessaloniki, Greece
[3] Bar Ilan University, Ramat Gan, Israel

Abstract. Runtime verification (RV) facilitates monitoring the executions of a system, comparing them against a formal specification. A main challenge is to keep the incremental complexity of updating its internal structure, each time a new event is inspected, to a minimum. There is a tradeoff between achieving a low incremental complexity and the expressive power of the used specification formalism. We present an efficient RV tool that allows specifying properties of executions that include data, with the possibility to apply arithmetic operations and comparisons on the data values. In order to be able to apply efficient RV for specifications with these capabilities, we combine two RV methodologies: the first one is capable of performing arithmetic operations and comparisons based on the most recent events; the second is capable of handling many events with data and relating events that occur at arbitrary distance in the observed execution. This is done by two phase RV, where the first phase, which monitors the input events directly and is responsible to the arithmetic calculations and comparisons, feeds the second phase with modified events for further processing. This is implemented as a tool called TP-DEJAVU, which extends the DEJAVU tool.

1 Introduction

Runtime verification (RV) allows monitoring the execution of systems, checking them against a formal specification. RV is often restricted to checking one execution at a time and viewing a prefix of the execution at any given moment, providing a verdict, which is often *true* or *false*. With these limitations, RV is devoid of some of the complexity and computability restrictions of more comprehensive formal methods such as formal verification and model checking [9].

Monitoring of executions with data against a first-order specification is challenging because of the need to keep and handle a large number of data values that appear in the observed prefix and the relationship between them. We are interested here in the capability of applying arithmetic operations and comparisons between data elements. In

* The research performed by the first author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research performed by the third and fourth authors was partially funded by Israeli Science Foundation grant 2454/23: "Validating and controlling software and hardware systems assisted by machine learning".

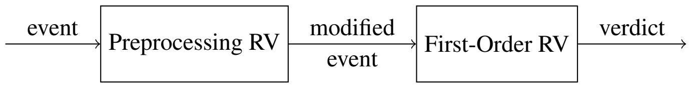

FIG. 1: Two phase monitoring

a general setting, a specification language should enable comparing different values that appear in distant events in the inspected execution. For example, checking that a new value that appears in the current event is exactly twice as big as some value observed in some previous event; this would require a comparison of the newly inspected value to the set of previously observed values. However, we have observed that in practice, in many cases the operations and comparisons required by the specification are limited, and can be performed using a fixed amount of memory. This includes in particular arithmetic operations that are based on past events within a limited distance, or on aggregate functions for calculating, e.g., sum, count, average, etc.

We introduce the TP-DEJAVU tool, for *Two Phase* DEJAVU. It combines two RV techniques: one that is capable of performing a rich collection of arithmetic, Boolean and string calculations, although restricted to using a fixed amount of memory elements, and based on the most recently seen or kept (through aggregation) values. The other one facilitates efficient processing of relational data but devoid of arithmetic operations and comparisons. This is done by applying a two phase RV processing (Figure 1), where the first phase, which we also call the *preprocessing*, is responsible for the arithmetic manipulations, and the second part implements first-order based checks. The two parts interact through events sent by the first phase to the second one. These events can be modified from the original monitored events observed by the first phase.

RV algorithms typically maintain a *summary* that contains enough information for deciding on future verdicts without having to keep the full observed prefix. The implementation typically keeps a set of variables and updates them upon inspecting a new event. One can classify two main dichotomies for specifications for RV. The first one is *operational* specification. There, the specification describes the dynamics of changing the summary performed based on inspected events. This can be done by means of a set of assignments to summary variables, which are updated each time a new event is intercepted.

The second dichotomy is *declarative* specification, where *constraints* on the monitored sequence of events are given. An example of a declarative specification formalism is propositional temporal logic. It can express the relation between Boolean values that represent some inspected or measured properties (e.g. the light is on, a communication was successful) at different (not necessarily adjacent) time points during the monitored execution. First-order temporal logic extends this to also allow relating data values occurring in different events, e.g., requiring that a value is read from a file only if that file was opened and not closed yet. RV for declarative specification typically involves a translation into an internal operational form, involving updates to the summary performed when a new event occurs.

For past time *propositional* LTL, there is a direct and efficient translation from the specification [22], where the truth value of each subformula of the specification, based on the currently inspected prefix, is represented using a Boolean variable in the summary. RV algorithms may require the use of more general objects in the summary than Boolean variables. These variables can be numeric, e.g., integers or (fixed precision) reals. The *incremental complexity* of updating the summary is an important factor to consider for online RV algorithms, since these updates need to keep up with the speed of appearing events.

When moving from propositional to *first-order* temporal logic, instead of simple variables, one can use *relations* that represent subformulas [20,6]; for a given subformula η of the given specification, such a relation represents all the assignments to its free variables that satisfy η. The RV tool DEJAVU uses a BDD encoding of relations that represent assignments that satisfy the subformulas of the specification. To achieve high efficiency in time and space, exploiting BDD compactness [7], the individual objects of these relations are *bitstrings* that encode *enumerations* of the observed values rather than the values themselves. However, arithmetic operations and comparisons need to be performed based on the observed values themselves.

An arithmetic comparison between values, e.g., $<$, is implemented in DEJAVU by updating a relation $R_<$ that represents $<$ where $(v,w) \in R_<$ iff $v < w$. The relation $R_<$ is updated each time a new value v appears. This is done by comparing v against all the previously seen values. Abstractly, this can be thought of as adding to $R_<$ the tuples (v,w) when $v < w$ and (w,v) when $w < v$, for values w observed so far in the trace. Since making this update depends on the amount of values observed so far, this update can be time consuming, affecting the incremental complexity: it can grow unbounded with the length of the observed prefix. Furthermore, the need to perform arithmetic calculations and comparisons may tame down some representation optimization.

Our solution is to combine two approaches: the operational one to do some limited amount of arithmetic calculations and comparisons that are restricted to a finite amount of data, and the declarative one that is performed based on the QTL formalism of DEJAVU. We observed through the industrial case studies in the H2020 EU project FOCETA, from which we borrow some examples, that this is a powerful approach. On the other hand, we point out that some specifications (see Property 4 below) cannot be expressed using this combination.

The following examples show the complication involved in achieving a comparison between values within first-order temporal logic (the formal semantics of the logic is in Section 2.1), and gives a hint on where an alternative specification, based on combining operational and declarative components, may be useful.

1. $\forall x\, ((p(x) \wedge x > 7) \rightarrow \exists y \diamondminus q(x,y))$. Here, the constraint $(x > 7)$ is imposed only on the *most recent* event. This can be checked by preprocessing the monitored event and checking the condition $x > 7$ as this event is intercepted.
2. $\forall x \forall y\, ((p(x) \wedge \ominus q(y) \wedge x < y) \rightarrow \diamondminus r(x,y))$. The constraint $(x < y)$ is between the values observed within the current and the previous events. Remembering the value of y from the previous event, one can easily make this comparison. Using a finite amount of memory for past values will also work when $\ominus$ is replaced with some fixed number of the $\ominus$ operator.
3. $\forall x\, (p(x) \rightarrow (\forall y\, (\ominus \diamondminus q(y) \rightarrow x > y) \wedge \exists z \ominus \diamondminus q(z)))$. This property formalizes that when an event of the form $p(x)$ occurs, the value of x is bigger than any value y

seen so far within an event of the form $q(y)$, and further, that at least one such a $q(z)$ event has occurred. A straightforward implementation, used in DEJAVU, builds a relation of tuples (x,y) where $x > y$, updating the relation as new values appear in events. A closer look reveals that we could have used a simpler way to monitor this property by keeping at each point the *maximal* value of y seen so far within $q(y)$ and comparing it with the new value x in a current event $p(x)$.

4. $\forall x\ (p(x) \rightarrow \exists y \exists z(\diamondminus\, q(y) \wedge \diamondminus\, q(z) \wedge y \neq z \wedge x = (y+z)/2))$. This case is difficult, since the property requires that a new p event has a value that is the average of two distinct values observed in previous q events. This requires remembering and comparing against all the previous values that appeared in previously observed q events. This case may necessitate comparison with an unboundedly growing number of past values. It cannot be handled by the particular two phase RV method suggested in this paper, which allows arithmetic computations limited to a finite amount of stored values.

As mentioned before, we split the specification into an operational part and a declarative part (cf. Figure 1). The operational part, which specifies arithmetic operations and comparisons, is handled as a preprocessing stage. The declarative part is restricted not to include arithmetic components. The RV is then combined from the preprocessing of the intercepted events by the operational part, which generates modified (augmented) events for the declarative specification. The latter is handled by the old version of DEJAVU. Without the capability of arithmetic comparisons, DEJAVU is an extremely efficient tool for processing traces with data, see the experiments in [20]; using our two-phase approach, we enhance its capabilities to perform with similar efficiency, albeit the added expressiveness.

Related work Some early tools supported data comparison and computations as part of the logic [2,4]. The version of MONPOLY tool in [5] supports comparisons and aggregate operations such as sum and maximum/minimum within a first-order LTL formalism. It uses a database-oriented implementation. Other tools supporting automata based limited first-order capabilities include [10,26]. In [13], a framework that lifts the monitor synthesis for a propositional temporal logic to a temporal logic over a first-order theory, is described. A number of internal DSLs (libraries in a programming language) for RV have been developed [11,16,18], which offer the full power of the host programming language for writing monitors and therefore allow for arbitrary comparisons and computations on data to be performed. The concept of phasing monitoring such that one phase produces output to another phase has been explored in other frameworks, including early frameworks such as [24] with a fixed number of phases, but with propositional monitoring, and later frameworks with arbitrary user defined phases [4,17]. In particular, stream processing systems support this idea [12,23,25]. A more remotely related work on increasing the expressive power of temporal logic is the extension of DEJAVU with rules described in [21].

2 Combining Operational and Declarative RV

The structure of RV algorithms is typically simple and consists of capturing a monitored event and updating its internal memory, which we call a *summary*, as it summarises the needed information from the observed prefix of events. In case that a verdict is available, it is reported, sometimes also causing the verification procedure to terminate, if this kind of verdict is stable for the given specification (e.g., a negative verdict for a safety property).

2.1 Declarative specification in past time first-order temporal logic

The logic QTL, used by DEJAVU, and a core subset of the logic used by the MONPOLY tool [6], is a 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 fully supports predicates over multiple arguments.

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

A formula can be interpreted over multiple types (domains), e.g., natural numbers or strings. Accordingly, each variable, constant and parameter of a predicate is defined over a specific type. 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 $\mathit{type}(x)$.

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

Semantics. A QTL formula is interpreted over a *trace*, 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 parameters belong to particular domains that are associated with (places in) the predicates. A more general semantics can allow each event to consist of a set of predicates with multiple parameters. However, this is *not* implemented in DEJAVU.

QTL subformulas have the following informal meaning: $p(a)$ is true if the last event in the trace is $p(a)$. The formula $p(x)$, for some variable x, holds if x is bound to a constant a such that $p(a)$ is the last event in the trace. The formula $(\varphi \, \mathcal{S} \, \psi)$, which reads as φ *since* ψ, means that ψ occurred in the past (including now) and since then (beyond that state) φ has been true. (The *since* operator is the past dual of the future time *until* modality in the commonly used future time temporal logic.) The property $\ominus \varphi$ means that φ is true in the trace that is obtained from the current one by omitting the last event. 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: $\mathit{false} = \neg\mathit{true}$, $(\varphi \lor \psi) = \neg(\neg\varphi \land \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \lor \psi)$, $\diamondminus \varphi = (\mathit{true} \, \mathcal{S} \, \varphi)$ ("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 $\eta \in \mathit{sub}(\varphi)$, i.e. η is a subformula of φ. 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 v. We also write $[v \mapsto a]$ to denote the assignment that consists of a single variable v mapped to value 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 $\gamma|_{free(\varphi)}$ 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 $free(\varphi)$, and $i \geq 1$.

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

2.2 RV monitoring first-order past LTL

We review the algorithm for monitoring first-order past LTL, implemented as part of DejaVu [20]. Consider a classical algorithm for past time propositional LTL [22]. For propositional LTL, an event is a set of propositions. 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. When a new event e occurs, the vector now is copied to the vector pre; then a new version of the vector now is calculated based on the vector pre and the event e as follows:

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

The *first-order* monitoring algorithm replaces the two vectors of bits by two vectors that represent *assignments*. The first vector, pre, contains, for each subformula η of the specification, a relation that represents the set of assignments to the free variables of η that satisfy η after the monitored trace seen so far *except its last event*. The second vector, now, contains the assignments that satisfy η given the complete monitored trace seen so far.

The updates in the first-order case replaces negation using complementation (denoted using an overline), conjunction with intersection $\cap$ and disjunction with union $\cup$. The intuition behind the connection of the Boolean operators for the propositional logic and the set operators for QTL follows from redefining the semantics of QTL in terms of sets. Let $I[\varphi, \sigma, i]$ be the function that returns the *set* of assignments such that $(\gamma, \sigma, i) \models \varphi$ iff $\gamma|_{free(\varphi)} \in I[\varphi, \sigma, i]$. Then, e.g., $I[(\varphi \wedge \psi), \sigma, i] = I[\varphi, \sigma, i] \bigcap I[\psi, \sigma, i]$. For other cases and further details of set semantics, see [20].

The complementation, intersection and union are operators on relations. Note that the relations for different subformulas can be over different (but typically not disjoint) types of tuples. For example, one relation can be formed from a subformula with free variables x, y, z and the other one over y, z, w, say with x, w over strings and y, z over integers. Intersection actually operates as a database *join*, which matches tuples of its two arguments when they have the same values for their common variables. Union is actually a *co-join*. The symbol $\perp$ represents the relation with no tuple, and $\top$ represents the relation with all possible tuples over the given domains.

In order to form a finite representation of an unbounded domain, one can represent the values that were already observed in previous events, plus a special notation for all the values not yet seen, say "$\sharp$". This representation is updated each time a new value is observed in an input event. For example, the set $\{\sharp, 4, 9\}$ represents all the values that we have *not* seen so far in input events, and in addition, the values 5 and 7. This finite representation allows using full negation in monitored formulas.

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

1. Initially, for each $\eta \in sub(\varphi)$ of the specification φ, $\mathsf{now}(\eta) = \perp$.
2. Observe a new event $p(a)$ as input;
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 $\mathsf{now}(\eta)$.
   - $\mathsf{now}(true) = \top$
   - $\mathsf{now}(p(a)) =$ if current event is $p(a)$ then $\top$ else $\perp$
   - $\mathsf{now}(p(x)) =$ if current event is $p(a)$ then $\{a\}$ else $\perp$
   - $\mathsf{now}((\eta \wedge \psi)) = (\mathsf{now}(\eta) \cap \mathsf{now}(\psi))$
   - $\mathsf{now}(\neg\eta) = \overline{\mathsf{now}(\eta)}$
   - $\mathsf{now}((\eta\, \mathcal{S}\, \psi)) = (\mathsf{now}(\psi) \cup (\mathsf{now}(\eta) \cap \mathsf{pre}((\eta\, \mathcal{S}\, \psi))))$
   - $\mathsf{now}(\ominus\, \eta) = \mathsf{pre}(\eta)$
   - $\mathsf{now}(\exists x\, \eta) = \mathsf{now}(\eta)_x$
5. Goto step 2.

In DEJAVU, for each value a seen for the first time in an input trace one assigns an enumeration. Using hashing, subsequent occurrences of the value a obtain the same enumeration. This value is then converted to a *bitstring*. For a subformula η of the specification, $\mathsf{now}(\eta)$ is a BDD representation of concatenations of such bitrstrings, corresponding to the tuples of values of the free variables of η that satisfy η. For further details on the DEJAVU implementation, see [20]. The DEJAVU tool uses keyword characters to express QTL formulas; it employs the following notation: forall and exists stand for $\forall$ and $\exists$, respectively, P, H, S and @ for $\diamondminus$, $\boxminus$, $\mathcal{S}$ and $\ominus$, respectively, and |, & and ! for $\vee$, $\wedge$ and $\neg$, respectively.

2.3 The operational RV

An operational specification can be expressed using updates to variables that are included in the summary. The next state, which is the updated summary, is obtained by performing the assignments based on the values of the previous summary and the parameters appearing within the new event. The input to the first phase of RV is, as in

DEJAVU, a sequence of events $e_1.e_2.e_3\ldots$, where each event consists of a predicate name with parameters, enclosed in parentheses and separated by commas, as in $q(a,7)$. The syntax of the specification consists of two parts:

Initialization. This part starts with the keyword initiate. It provides values to the variables in the *initial* summary, i.e., the summary *before* any event occurs.

Update. Depending on the input predicate p, the variables of the summary are updated based on the values that are associated with the current event and the previous values of the variables in the previous summary, as described below. An updated event is then generated.

An update consists of a sequence of on clauses, starting with "on $p(x,y,\ldots)$", where p is a predicate name and $x,y,\ldots$ are variables that are set to the positional arguments of p. Thus, for the event $p(3,$"*abc*"$)$, $p(x,y)$ will cause x to be assigned to 3 and y to "*abc*". Such on *clause* is followed by a sequence of assignments to be executed when an event that matches the on clause occurs. The assignments can include Boolean operations, arithmetic operations (e.g., $*$, $/$, $+$, $-$), arithmetic comparisons ($<$, $\leq$, $>$, $\geq$) and string operations.

The construct *ite* stands for *if-then-else*. Its first parameter is a Boolean expression (obtained, e.g., from an arithmetic comparison). The second parameter corresponds to the value returned when the Boolean expression calculates to *true* (the *then* case). The third parameter is the value returned when the Boolean expression calculates to *false* (the *else* case). The second and third parameters must be of the same type. This can be also extended to string operators. The symbol @ prefixing a variable x means that we refer to the value of x in the *previous* summary. Thus, $x := ite(v > @x, v, @x)$, where v is a value that appears in the current event, updates x to keep the maximal value of v that was observed. If the value of $@x$ is needed for the first event then x must be initialized. If a summary variable is not assigned to, in the on clause that matches the current event, then this variable retains its value from the previous summary. This helps to shorten the description of the property.

The operational phase can deal with different types; the types of variables are declared when they are assigned. We do not necessarily need to provide an on clause for each predicate that can appear in an event; if some predicate is not intercepted by an on clause in the operational phase, the event is forwarded unchanged to the declarative phase.

Example 1. In this example we have events that include two data items: a car vendor and speed. A *true* verdict is returned each time that a new vendor has broken the speed record *for the first time*. Note that the verdict for the inspected traces can change from *true* to *false* and back to *true* any number of times (the *false* verdict is *not* necessarily stable, as is the case for safety properties [1]).

```
 initiate
    MaxSpeed: int := 0
on recorded(vendor: str, speed: int)
    NewRecord: bool := @MaxSpeed <speed
    MaxSpeed: int := ite (NewRecord == true, speed, @MaxSpeed)
    output fast(vendor, NewRecord)
```

```
exists x . (fast(x, "true") & ! @ P fast(x, "true"))
```

Example 2. The following example records from time to time the temperature in one of a collection of running cars. It calculates the temperature increase between the two successive startMeasure and endMeasure events. If the increase in temperature is more than 5 degrees (say Celsius), then it indicates an overheated car. The monitor reports if some car is overheated more than twice.

```
on startMeasure(Car: str, temp: float)
    output startMeasure(Car)
on endMeasure(Car: str, temp: float)
    Warming: bool := (temp − ⊖temp) > 5
    output warmAlert(Car, Warming)
exists x . (warmAlert(x, "true") & @(startMeasure(x) & P (warmAlert(x, "true") &
@P (warmAlert(x, "true")))))
```

Example 3. The next property deals with temperature setting commands sent to air conditioners (ac). In order to send a command, we must have turned the air conditioner on (and not off since). However, if the command is out of temperature bounds, specifically, below 17C or above 26C, then it is ignored as a faulty command, and there is no such requirement.

```
on set(ac: str, temp: float)
    WithinBound: bool := (temp >= 17 && temp <= 26)
    output set(ac, temp, WithinBound)
forall ac . ((exists temp . set(ac, temp, "true")) ->(!turn_off(ac) S turn_on(ac)))
```

2.4 Examples from Use Cases of Autonomous Systems

In this subsection, we detail specifications developed for learning-enabled autonomous systems (LEAS) as part of the H2020 European Union FOCETA project.

Prediction of obstacle behaviour in an automated valet parking system An automated valet parking (AVP) system is an L4 (level four) autonomous driving system. A user owning a vehicle equipped with the AVP functionality can stop the car in front of the parking lot entry area. Whenever the user triggers the AVP function, the vehicle will communicate with the infrastructure and park the car at designated regions. The system is expected to operate under mixed traffic, i.e., the parking lot will have other road users including walking pedestrians and other vehicles.

The AVP system implements object detection capabilities for sensing the surrounding objects and localisation functions for inferring the system's location on the map. It also features mission and path planning functions, as well as trajectory tracking and a prediction module that is based on the available information that predicts positions of traffic participants (i.e., obstacle list) in the future. The position is given in x, y coordinates. The prediction error is computed as the distance between an obstacle's actual position at cycle t and the predicted position for it at cycle $t-1$. The prediction error reported for an obstacle at any computation cycle is bounded within a certain value Epo. Also, a maximum accumulated error Emax exists for the system.

```
initiate
    LErr:double := 0
    SysErr:double := 0
    NewPred:bool := false
on mk_prediction(ru:str, px:int, py:int)
    LErr:double := 0
    NewPred:bool := true
    output predicted(ru)
on obstacle(ru:str, x:int, y:int)
    LErr:double := ite(@NewPred == true, (((x−@px)^2)+((y−@py)^2)^0.5), 0)
    SysErr:double := @SysErr+LErr
    error:bool := (LErr > Epo) || (SysErr > Emax)
    NewPred:bool := false
    output valid(ru,error,L_err)
on exit(ru:str,p_err:double)
    SysErr: double := @SysErr − p_err
    output exit(ru)

forall ru . (exit(ru) | ((@predicted(ru) -> exists n . valid(ru,"false", n)) S entry(ru))
```

This specification states that a prediction can be made for the future position of any road user (ru) that has entered the parking lot area (event entry) and has not yet exited (event exit). Upon having a new prediction (NewPred), the prediction error (LErr) measured when ru's actual position is known (event obstacle) does not exceed Epo. Moreover, the accumulated error (SysErr) for all road users that are still moving in the parking lot area does not exceed Emax. When a ru leaves the parking space, the last reported prediction error (p_err) ceases to be considered in the accumulated error (SysErr).

Monitoring the perception function of an AVP system The perception monitor takes as input from the perception subsystem the computed free space area (event new_-fspace), obstacles (ru) and their localisation (event location). The dimensions of the free space are given as a rectangle defined by its diagonally opposite corners. The dimensions of localised rus are given through reporting their detected 2D bounding boxes. The perception monitor detects and alerts about inconsistencies in the perception process. Specifically, the output of free space detection and localisation of rus should never overlap, to ensure consistency of the overall perception function.

```
on new_fspace(fs_x1: int, fs_y1: int, fs_x2: int, fs_y2: int)
    output new_fspace(fs_x1, fs_y1, fs_x2, fs_y2)
on location(ru:str, bb_xa: int, bb_ya: int, bb_xb: int, bb_yb: int)
    overlap:bool := (bb_xb >= fs_x1) && (bb_xa <= fs_x2)
    && (bb_yb <= fs_y1) && (bb_ya >= fs_y2)
    output error(ru,overlap)
on exit_fspace(fs_x1: int, fs_y1: int, fs_x2: int, fs_y2: int)
    output exit_fspace(fs_x1, fs_y1, fs_x2, fs_y2)

exists x1 . exists y1 . exists x2 . exists y2 . ( exit_fspace(x1,y1,x2,y2) |
 (forall ru . !error(ru, "true") & (!exit_fspace(x1,y1,x2,y2) S new_fspace(x1, y1, x2, y2)))
```

3 The TP-DEJAVU Tool and Experimental Results

TP-DEJAVU is an extension of DEJAVU [29], both written in Scala. It facilitates two phase RV processing. The first phase (the prepossessing) performs operational RV with syntax as described in Section 2.3. Upon each intercepted event, it performs summary update calculations and generates a modified event, which it passes to the declarative phase tool. The second phase is based on the DEJAVU tool, and performs monitoring against a first-order specification. The source of the TP-DEJAVU tool, with documentation and experiment files, including the examples described in this paper, appear in [28].

Next, we present experimental results on the relative efficiency of runtime monitoring QTL specifications using DEJAVU tool versus the efficiency of two phase RV using TP-DEJAVU. In all cases, the two specifications output exactly the same results. Our benchmarks focused exclusively on the time and memory consumption during the evaluation phase, without the compilation time. The properties in our experiment were taken from Section 1, specifically properties 1-3 (property 4 cannot be translated into TP-DEJAVU); their adaptations for TP-DEJAVU are illustrated in Figure 2.

Traces: For properties 1 and 2, the event sequences and their respective values were assigned at random. However, for property 3, we tried to construct traces such that DEJAVU can process them within the given time limit of 1000 seconds. Out of every 100 events, one event is labelled as “p”, while the rest are labelled “q”. For our experiment, we utilized traces comprising 10K, 100K, 500K, 1M, and 5M events.

Results Table 1 presents the results from our comparison experiments. These results highlight the advantages of a two phase RV over the singular phase launched by DEJAVU when complex inequality operators are involved. Executions where the evaluation process exceeded 1000 seconds are marked with the symbol ∞. For property 1, both methods display a rise in execution times with increasing trace sizes. In this case, TP-DEJAVU manages to be marginally faster but may need more memory. For Property 2 and 3, DEJAVU was unable to evaluate any of the traces (except one), as evidenced by the consistently infinite execution times. On the other hand, TP-DEJAVU managed to assess all traces successfully. However, this came with a significant increase in memory consumption, especially evident with the larger trace files.

```
on p(x: int)
  in_bound: bool := x > 7
  output p(x, in_bound)

forall x . (p(x, "true") ->
    exists y q(x,y))
```

```
initiate
  prev_q: bool := false

on p(x: int)
  x_lt_y: bool := x < y
  prev_q: bool := false
  output p(x, x_lt_y)

on q(y: int)
  prev_q: bool := true
  output q(y)

forall x . forall y .
  ((p(x, "true") & @q(y)) ->
  P r(x, y))
```

```
initiate
    MaxY: int := −1

on p(x: int)
 xGTy: bool := x > MaxY
 output p(x, xGTy)

on q(y: int)
 NewMaxY: bool := @MaxY < y
 MaxY: int :=
 ite (NewMaxY, y, @MaxY)
 output q(y)

forall x . forall w .
  (p(x, w) -> (p(x, "true") &
  exists y . @P q(y)))
```

FIG. 2: TP-DEJAVU properties corresponding to properties 1 (left), 2 (middle), and 3 (right).

Property	Method	Trace $10K$	Trace $100K$	Trace $500K$	Trace $1M$	Trace $5M$
$P1$	DEJAVU	0.64s 125.26MB	1.31s 335.52MB	4.76s 1.10GB	8.85s 1.88GB	185.65s 3.59GB
	TP-DEJAVU	0.54 129.74MB	0.96s 311.24MB	3.25s 858.30MB	5.44s 1.21GB	41.18s 5.70GB
$P2$	DEJAVU	∞ -	∞ -	∞ -	∞ -	∞ -
	TP-DEJAVU	0.56s 135.18MB	1.12s 308.72MB	4.12s 1.17GB	7.54s 1.78GB	56.78s 3.55GB
$P3$	DEJAVU	5.23s 805.47MB	∞ -	∞ -	∞ -	∞ -
	TP-DEJAVU	0.64s 119.85MB	0.90s 326.38MB	2.39s 738.78MB	4.08s 1.11GB	21.30s 3.43GB

TABLE 1: TP-DEJAVU vs. DEJAVU.

In Table 2, we provide experimental results highlighting the efficiency of the two-phase RV based on the examples detailed in Sections 2.3 and 2.4. This includes Examples 1 to 3 and the two LEAS applications.

4 Conclusions and Further Work

We presented an RV approach based on two phases. The first, operational RV phase, takes care of algebraic (but also Boolean and string) calculations. The second, declarative RV phase, takes care of relational computation. The experimental results indicate that the two phase approach can gain considerable advantage in speed over incrementally encoding relations that represent comparisons between the observed data by the declarative RV, as is done in DEJAVU. It allows handling a rich number of algebraic operators and comparisons. On the other hand, it is limited to applying these operations to a fixed number of kept values, recently observed or aggregated from earlier events. An extension that allows storing unbounded collections of values can be considered. Currently, the operational phase of the TP-DEJAVU tool outputs a single event for every input event. An extension of the tool can relax this rule and can generate several or zero events, or even have several possibilities, depending on the data in the events.

Property	# Objects	Trace $10K$	Trace $100K$	Trace $500K$	Trace $1M$	Trace $5M$
Example 1	10 cars	0.53s 105.68MB	0.92s 157.34MB	1.55s 229.31MB	2.35s 292.72MB	5.74s 303.02MB
Example 2	10 cars	0.67s 106.88MB	0.92s 150.04MB	1.52s 263.37MB	2.41s 288.28MB	11.12s 291.95MB
Example 3	10 ACs	0.58s 108.30MB	0.81s 165.85MB	1.68s 240.72MB	2.28s 297.82MB	7.22s 300.19MB
Use case 1	10 rus	0.72s 114.17MB	1.54s 169.08MB	1.95s 292.89MB	3.23s 296.25MB	8.81s 315.48MB
Use case 2	10 rus	0.64s 134.00MB	1.30s 189.15MB	3.37s 340.44MB	6.10s 349.80MB	22.72s 1.11GB

TABLE 2: Examples and use cases of Two Phase RV

Data-Availability Statement

The TP-DejaVu tool is open source and publicly available at `https://doi.org/10.5281/zenodo.8322559`, as well as from the GitHub repository at `https://github.com/moraneus/TP-DejaVu`.

References

1. B. Alpern, F. B. Schneider, Recognizing Safety and Liveness. Distributed Computing 2(3), 117-126, 1987.
2. H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-Based Runtime Verification, Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS 2937, 44-57, Springer, 2004.
3. H. Barringer, K. Havelund, TraceContract: A Scala DSL for Trace Analysis, Proc. of the 17th International Symposium on Formal Methods (FM'11), LNCS 6664, 57-72, Springer, 2011.
4. H. Barringer, D. Rydeheard, K. Havelund, Rule Systems for Run-Time Monitoring: from Eagle to RuleR, Seventh Workshop on Runtime Verification (RV), LNCS 4839, 111-125, Springer, 2007.
5. D. A. Basin, F. Klaedtke, Sr. Marinovic, E. Zalinescu, Monitoring of temporal first-order properties with aggregations. Formal Methods Syst. Des. 46(3): 262-285, 2015.
6. D. A. Basin, F. Klaedtke, S. Müller, E. Zalinescu, Monitoring Metric First-Order Temporal Properties, Journal of the ACM 62(2), 45, 2015.
7. R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Comput. Surv. 24(3), 293-318, 1992.
8. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, Symbolic Model Checking: 10^{20} States and Beyond, LICS, 1990, 428-439.
9. E. M. Clarke, O. Grumberg, D. A. Peled, Model checking, 1st Edition. MIT Press 2001, ISBN 978-0-262-03270-4, pp. I-XIV, 1-314
10. C. Colombo, A. Gauci, G. J. Pace, LarvaStat: Monitoring of Statistical Properties. RV 2010: 480-484
11. D. Dams, K. Havelund, S. Kauffman, A Python Library for Trace Analysis, 22nd International Conference on Runtime Verification (RV), LNCS 13498, 264–273, Springer, 2022.
12. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, Z. Manna: LOLA: Runtime Monitoring of Synchronous Systems, 12th International Symposium on Temporal Representation and Reasoning (TIME), 2005, 166-174, IEEE.
13. N. Decker, M. Leucker, D. Thoma, Monitoring modulo theories, Software Tools for Technology Transfer (STTT), 18(2), 205-225, 2016.
14. B. Duckett, K. Havelund, L. Stewart, Space Telemetry Analysis with PyContract, Applicable Formal Methods for Safe Industrial Products - Essays Dedicated to Jan Peleska on the Occasion of His 65th Birthday, LNCS 14165, 264–273, Springer, 2023.
15. M. Fowler, R. Parsons, Domain-Specific Languages, Addison-Wesley, 2010
16. F. Gorostiaga, C. Sánchez, H. Striver: A Very Functional Extensible Tool for the Runtime Verification of Real-Time Event Streams, Formal Methods, LNCS 13047, 563-580, Springer, 2021.
17. S. Hallé, R. Villemaire, Runtime Enforcement of Web Service Message Contracts with Data, IEEE Transactions on Services Computing, Volume 5 Number 2, 192-206, 2012.
18. K. Havelund, Data Automata in Scala, Theoretical Aspects of Software Engineering Conference (TASE), 1-9, IEEE Computer Society, 2014.

19. K. Havelund, Rule-based runtime verification revisited, Software Tools for Technology Transfer (STTT), 17(2), 143-170, Springer, 2015.
20. K. Havelund, D. Peled, D. Ulus, First-order Temporal Logic Monitoring with BDDs. FM-CAD 2017, 116-123.
21. K. Havelund, D. Peled, An extension of first-order LTL with rules with application to runtime verification. Int. J. Softw. Tools Technol. Transf. 23(4): 547-563, 2021.
22. K. Havelund, G. Rosu, Synthesizing Monitors for Safety Properties, TACAS'02, LNCS Volume 2280, Springer, 342-356, 2002.
23. H. Kallwies, M. Leucker, M. Schmitz, A. Schulz, D. Thoma, A. Weiss, TeSSLa - An Ecosystem for Runtime Verification, 22nd International Conference on Runtime Verification (RV), LNCS 13498, 314-324, Springer, 2022
24. M. Kim, S. Kannan, I. Lee, O. Sokolsky, Java-MaC: a Run-time Assurance Tool for Java, 1st Int. Workshop on Runtime Verification (RV), ENTCS 55(2), Elsevier, 2001.
25. I. Perez, F. Dedden, A. Goodloe, Copilot 3, Technical report, NASA Langley Research Center, 2020.
26. G. Reger, H. C. Cruz, D. Rydeheard, MarQ: Monitoring at Runtime with QEA, 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 9035, 596-610, Springer, 2015.
27. P. Wolper, Temporal logic can be more expressive, Information and Control 56(1/2): 72-99, 1983.
28. TP-DEJAVU tool source code, `https://doi.org/10.5281/zenodo.8322559`.
29. DEJAVU tool source code, `https://github.com/havelund/dejavu`.