Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors

Howard Barringer[1], Yliès Falcone[2], Klaus Havelund[3] ⋆,
Giles Reger[1]⋆⋆, and David Rydeheard[1]

[1] University of Manchester, UK
[2] University of Grenoble, France
[3] Jet Propulsion Laboratory, California Inst. of Technology, USA

Abstract. Runtime verification is the process of checking a property on a trace of events produced by the execution of a computational system. Runtime verification techniques have recently focused on parametric specifications where events take data values as parameters. These techniques exist on a spectrum inhabited by both efficient and expressive techniques. These characteristics are usually shown to be conflicting - in state-of-the-art solutions, efficiency is obtained at the cost of loss of expressiveness and vice-versa. To seek a solution to this conflict we explore a new point on the spectrum by defining an alternative runtime verification approach. We introduce a new formalism for concisely capturing expressive specifications with parameters. Our technique is more expressive than the currently most efficient techniques while at the same time allowing for optimizations.

1 Introduction

Runtime Verification [1–5, 7–11] is the process of checking a property on a trace of events produced by the execution of a computational system. Over the last decade, a number of different formalisms were proposed for specifying such properties and mechanisms for checking traces. Early work focused on *propositional* events but recently there has been a growing interest in so-called *parametric* properties where events carry data values. Challenges that arise when designing a runtime verification framework incorporating parametric properties are twofold. The first lies in the (parametric) specification formalism used to specify the property; usually one seeks expressiveness. The second lies in the efficiency of monitoring algorithms associated with the formalism.

A spectrum of runtime verification. Specification formalisms differ in their level of expressiveness and usability and, monitoring algorithms differ in efficiency. In developing monitoring frameworks, one can distinguish between systems such as JAVAMOP [10] and TRACEMATCHES [1], which focus on efficiency rather than expressiveness, and systems such as EAGLE [2], RULER [2, 5], LOGSCOPE [3] and TRACECONTRACT [4], which focus on expressiveness rather than efficiency. The development in this paper arose from our attempt to understand, reformulate and generalise parametric trace slicing (as adopted by JAVAMOP [7]), and more generally from our attempt to explore the spectrum between JAVAMOP and more expressive systems such as EAGLE, RULER, LOGSCOPE and TRACECONTRACT.

⋆ Part of the research described in this publication was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

⋆⋆ The work of this author was supported by the Engineering and Physical Sciences Research Council [grant number EP/P505208/1].

Contributions. This paper contributes to the general effort to understand the spectrum of monitoring techniques for parametric properties. We propose Quantified Event Automata (QEA) as a formalism for defining parametric properties that is more expressive than the formalisms behind the current most efficient frameworks such as JAVAMOP and TRACEMATCHES. This formalism is as expressive as the formalisms behind the most expressive frameworks, such as RULER, but is, in our opinion, more intuitive and allows for optimisation. Additionally we include guards and assignments in our new formalism. We present both a big-step semantics, operating on full finite traces, and a small-step semantics, operating on the trace step-by-step. The small-step semantics acts as a basis from which monitoring algorithms can be derived.

Paper Organization. Section 2 motivates our approach by exhibiting the limitations of parametric trace slicing and overviews how we overcome them. We introduce QEA in Sec. 4 by first defining Event Automata (EA) in Sec. 3. An EA defines a property over a set of parametric events, and QEA generalise these by quantifying over some variables in the EA. As we separate quantifications from the definition of the property we could replace Event Automata with some other formalism, such as context-free grammars, in the future. Sections 3 and 4 are concerned with the a big-step semantics of our formalism, whereas Sec. 5 presents a small-step semantics, along with a notion of acceptance in a four-valued verdict domain. Finally, we discuss related work in Sec. 6 and draw conclusions in Sec. 7.

2 Background

Runtime monitoring is the process of checking a property on a trace (finite sequence) of events. In this context, an event records some action or snapshot from the monitored system. A property defines a language over events and a monitor is a decision procedure for the property. An event is said to be *propositional* if it consists of a simple name, e.g., `open`, and *parametric* if it contains data values, e.g., `open`('file42'). We name properties and monitors in a similar way: propositional and parametric monitors, respectively.

A previous approach to parametric runtime monitoring is called parametric trace slicing [7] (an approach taken by JAVAMOP [10]). Here a parametric monitor, from a theoretical point of view, works by slicing its parametric input trace to a set of propositional traces that are then processed by separate propositional monitors. Let us illustrate this approach with a simple example. Consider the parametric property stating that for any file f, $\texttt{open}(f)$ and $\texttt{close}(f)$ events for that file f should alternate. This property can be formalised as the parametric regular expression $(\texttt{open}(f).\texttt{close}(f))^*$. Consider now the parametric trace $\texttt{open}(1).\texttt{close}(2).\texttt{close}(1)$. In this parametric trace there are two different instantiations of f, namely f=1 and f=2. In this case slicing produces the following configuration consisting of two bindings associated with propositional traces:

$$[f \mapsto 1] : \texttt{open.close} \qquad [f \mapsto 2] : \texttt{close}$$

Each of these traces are then monitored by the monitor corresponding to the propositional property $(\texttt{open.close})^*$. It is clear that for $[f \mapsto 2]$ the property does not hold.

For practical purposes, instead of mapping each binding to a propositional trace as above, a configuration instead maps the binding to a propositional monitor state, the state the monitor will be in after observing that propositional trace. When a monitor receives an event it combines the event's parameters with the variables associated with

that event to construct a binding (a map from variables to concrete values), and looks up the appropriate propositional monitor state for that binding, and then applies the propositional event in that monitor state to obtain a new state. For example, given the above trace, the first event $\texttt{open}(1)$ would be used to construct the binding $[f \mapsto 1]$. However, note that the binding constructed from an event does not necessarily match exactly any of the bindings in the configuration. Instead, a monitor state is updated if it is mapped to by any binding that *includes* the binding produced by the event. Looking up monitor states directly from events makes a slicing approach efficient.

The following definition defines for a given trace and a given binding what propositional trace this binding is mapped to, namely the slice corresponding to that binding.

Definition 1 (Parametric Trace Slicing). *Given a trace of parameterised events τ and a binding θ, the θ-slice of τ, written $\tau \downarrow_\theta$, is the propositional trace defined by:*

$$\epsilon \downarrow_\theta = \epsilon \qquad\qquad e(\theta').\tau \downarrow_\theta = \begin{cases} e.(\tau \downarrow_\theta) & \text{if } \theta' \sqsubseteq \theta \\ \tau \downarrow_\theta & \text{otherwise} \end{cases}$$

where ϵ is the empty trace, each parameterised event $e(\theta')$ consists of an event name e and a binding θ', and $\sqsubseteq$ is the submap relation on bindings.

However, as we shall see, parametric trace slicing has two main shortcomings. First, it is not possible to write a property where an event name is associated with two different lists of variables, for example $\texttt{open}(f)$ and $\texttt{open}(g)$, as when observing an event, such as $\texttt{open}(1)$, it must be possible to construct a unique binding, such as $[f \mapsto 1]$, hence relying on only one unique variable associated with `open` (in this case f). Second, the theory assumes that all variables take part in slicing - forcing their values to remain fixed w.r.t. a monitor. Third, the theory implicitly assumes universal quantification on all parameters, hence forbidding alternation with existential quantification. Below are some properties that are not expressible in this parametric trace slicing setting:

Talking Philosophers - Any two philosophers may not speak at the same time - if one starts talking another cannot start until the first stops. Given any philosophers x and y, the property must therefore differentiate between events $\texttt{start}(x)$ and $\texttt{start}(y)$.

Auction Bidding - Amounts bid for an item should be strictly increasing. If bidding is captured by the event $\texttt{bid}(item, amount)$ the value given to $item$ should be fixed w.r.t. a monitor, but the value given to $amount$ should be allowed to vary.

Candidate Selection - For every voter there must exist a party that the voter is a member of, and the voter must rank all candidates for that party.

Our more general formalism allows us to express these and other properties with additional new features. We first introduce Event Automata (EA) to describe a property with parametric events containing both values and variables. An event name can occur with different (lists of) parameters, for example $\texttt{start}(1)$, $\texttt{start}(x)$ and $\texttt{start}(y)$. We then introduce Quantified Event Automata (QEA), which generalise EA by quantifying over some of the variables, making them bound. Variables that are not quantified over, hence free, can be rebound as a trace is analysed. This is useful for specification purposes as we shall see. As we will always instantiate Event Automata before using them we can treat all variables as free variables and rebind them where necessary. In theory, trace acceptance can be decided using a set of instantiated EA generated using the QEA as a template and replacing quantified variables with values from their domain. In practice this approach is inefficient and we present an alternative that allows for optimisation.

3 Event Automata

An Event Automaton is a non-deterministic finite-state automaton whose alphabet consists of parametric events and whose transitions may be labelled with guards and assignments. These are generalised in the next section by quantifying over zero or more variables appearing in parametric events. Here we assume the Event Automaton has been instantiated and all quantified variables replaced with values.

We begin by formalising the structure of Event Automata, then give a transition semantics and define an Event Automaton's language, finishing with three examples.

We use $\overline{s}$ to denote a tuple $\langle s_0, \ldots, s_k \rangle$. We use $X \to Y$ and $X \rightharpoonup Y$ to denote sets of total and partial functions between X and Y, respectively. We write maps (partial functions) as $[x_0 \mapsto v_0, \ldots, x_i \mapsto v_i]$ and the empty map as $[\,]$. Given two maps A and B, the map override operator is defined as:

$$(A \dagger B)(x) = \begin{cases} B(x) & \text{if } x \in \underline{dom}(B), \\ A(x) & \text{if } x \notin \underline{dom}(B) \text{ and } x \in \underline{dom}(A), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

3.1 Syntax

We build the syntax from a set of propositional event names Σ, a set of values Val[4], and a set of variables Var (disjoint from Val) as follows.

Definition 2 (Symbols, Events, Alphabets and Traces). *Let $Sym = Val \cup Var$ be the set of all symbols (variables or values). An event is a pair $\langle e, \overline{s} \rangle \in \Sigma \times Sym^*$, written $e(\overline{s})$. An event $e(\overline{s})$ is ground if $\overline{s} \in Val^*$. Let $Event$ be the set of all events and $GEvent$ be the set of all ground events. A* trace *is a finite sequence of ground events. Let $Trace = GEvent^*$ be the set of all traces.*

We use x, y to refer to variables, s to refer to symbols, $\mathbf{a}$ to refer to ground events , $\mathbf{b}$ to refer to events which are not necessarily ground, and σ, τ to refer to traces. Note that we focus on finite traces. A continuously evolving system could be monitored through snapshots of finite traces - the trace seen so far.

Bindings are maps from variables to values, i.e., elements of $Bind = Var \rightharpoonup Val$. There is a partial order $\sqsubseteq$ on bindings such that $\theta_1 \sqsubseteq \theta_2$ iff θ_1 is a submap of θ_2. *Guards* are predicates on bindings, i.e., total functions in $Guard = Bind \to \mathbb{B}$. We use θ and φ to denote bindings and g to denote guards. A binding can be applied to a symbol as a substitution – replacing the symbol if it is defined in the binding. This can be lifted to events and used to give a definition of a ground event and an event matching:

Definition 3 (Substitution). *The binding $\theta = [x_0 \mapsto v_0, \ldots, x_i \mapsto v_i]$ can be applied to a symbol s and to an event $e(\overline{s})$ as follows:*

$$s(\theta) = \begin{cases} \theta(s) \text{ if } s \in \underline{dom}(\theta) \\ s \quad \text{otherwise} \end{cases} \qquad e\langle s_0, \ldots, s_j \rangle(\theta) = e\langle s_0(\theta), \ldots, s_j(\theta) \rangle$$

Definition 4 (Matching). *Given a ground event $\mathbf{a}$ an event $\mathbf{b}$, the predicate $\texttt{matches}(\mathbf{a}, \mathbf{b})$ holds iff there exists a binding θ s.t. $\mathbf{b}(\theta) = \mathbf{a}$. Moreover, let $\texttt{match}(\mathbf{a}, \mathbf{b})$ denote the smallest such binding w.r.t $\sqsubseteq$ if it exists (and is undefined otherwise).*

[4] For example, integers, strings or objects from an Object Oriented programming language.

Assignments are total functions on bindings, i.e., elements of $Assign = Bind \to Bind$. We use γ to denote assignments. Guards and assignments may be described in suitable languages. We do not need to specify particular languages, but will use standard programming language notation in examples and assume that assignments maintain values they do not explicitly update. Now we are in a position to define Event Automata (EA).

Definition 5 (Event Automat a). *An EA $\langle Q, \mathcal{A}, \delta, q_0, F\rangle$ is a tuple where Q is a finite set of states, $\mathcal{A} \subseteq Event$ is a finite alphabet, $\delta \in (Q \times \mathcal{A} \times Guard \times Assign \times Q)$ is a finite set of transitions, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states.*

3.2 Semantics

We give the semantics of EA within the context of an EA $\mathsf{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$.

Definition 6 (Configurations and Transition Relation). *We define* configurations *as elements of the set $Config = Q \times Bind$. Let $\to \subseteq Config \times GEvent \times Config$ be a relation on configurations s.t. configurations $\langle q, \varphi\rangle$ and $\langle q', \varphi'\rangle$ are related by the ground event $\mathbf{a}$, written $\langle q, \varphi\rangle \xrightarrow{\mathbf{a}} \langle q', \varphi'\rangle$, if and only if*

$$\exists \mathbf{b} \in \mathcal{A}, \exists g \in Guard, \exists \gamma \in Assign : (q, \mathbf{b}, g, \gamma, q') \in \delta \wedge \\ \texttt{matches}(\mathbf{a}, \mathbf{b}) \wedge g(\varphi \dagger \texttt{match}(\mathbf{a}, \mathbf{b})) \wedge \varphi' = \gamma(\varphi \dagger \texttt{match}(\mathbf{a}, \mathbf{b})).$$

Let the transition relation $\to_{\mathsf{E}}$ be the smallest relation containing $\to$ such that for any event $\mathbf{a}$ and configuration c if $\nexists c' : c \xrightarrow{\mathbf{a}} c'$ then $c \xrightarrow{\mathbf{a}}_{\mathsf{E}} c$.

The relation $\to_{\mathsf{E}}$ is lifted to traces. For any two configurations c and c', $c \xrightarrow{\epsilon}_{\mathsf{E}} c$ always holds, and $c \xrightarrow{\mathbf{a}.\tau}_{\mathsf{E}} c'$ holds iff there exists a c' such that $c \xrightarrow{\mathbf{a}}_{\mathsf{E}} c''$ and $c'' \xrightarrow{\tau}_{\mathsf{E}} c'$.

In an EA, a configuration contains the values bound to the variables. These bindings are local to each EA – notably there is no shared global state. A ground event $\mathbf{a}$ can take $\langle q, \varphi\rangle$ into $\langle q', \varphi'\rangle$ if there exists a transition in δ starting in q, s.t. the events match, the guard is satisfied, and the new configuration contains the binding given by the assignment and state q'. Note that EA are non-deterministic.

Note. When we encounter an event for which there is no matching transition in the automaton we wait in the current state. There are alternative accounts, which are equivalent in the sense that we can translate automata between the different semantics. We have made this choice as our initial experience is that this makes writing specifications more straightforward as, when defining a property, we need not write transitions for events which have no role in the specification.

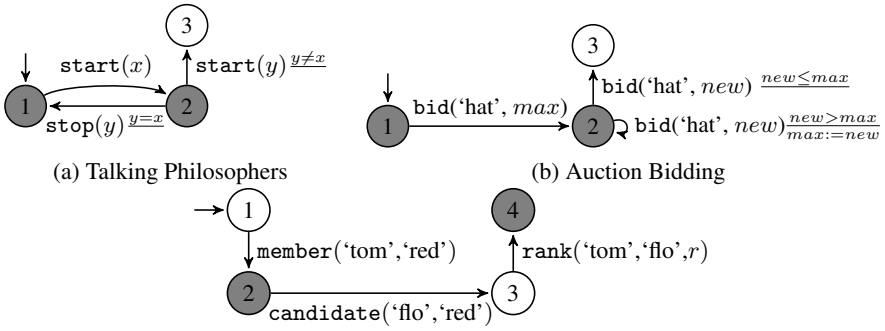
Let us now define the language of an EA. An event denotes a set of ground events – for example, the event $\texttt{start}(x)$ denotes the set $\{\texttt{start}(v) \mid v \in Val\}$ and a ground event denotes the singleton set containing itself. We use this notion to define the *ground alphabet* of an EA. Let the ground alphabet of the EA E be

$$\texttt{ground}(\mathsf{E}) = \{\mathbf{a} \in GEvent \mid \exists \mathbf{b} \in \mathcal{A} : \texttt{matches}(\mathbf{a}, \mathbf{b})\}.$$

We say that there is a run on τ reaching a configuration c iff $\langle q_0, [\,]\rangle \xrightarrow{\tau}_{\mathsf{E}} c$. An EA accepts a trace if there is a run on that trace reaching a configuration in a final state.

Definition 7 (Event Automaton Language). *The language of the Event Automaton E is noted and defined as*

$$\mathcal{L}(\mathsf{E}) = \{\tau \in \texttt{ground}(\mathsf{E})^* \mid \exists \langle \mathsf{q}, \varphi\rangle \in Config : \langle \mathsf{q}_0, [\,]\rangle \xrightarrow{\tau}_{\mathsf{E}} \langle \mathsf{q}, \varphi\rangle \wedge \mathsf{q} \in F\}.$$



(a) Talking Philosophers (b) Auction Bidding

(c) Candidate Selection

Fig. 1: Three EAs. We use shaded states to indicate final states and the notation $\frac{guard}{assignment}$ for writing guards and assignments on transitions.

3.3 Examples

To illustrate Event Automata and their languages, consider the three examples in Sec. 2, Talking Philosophers, Auction Bidding and Candidate Selection. Recall that all variables occuring in an EA are unquantified (free).

Talking Philosophers. The EA Phil in Fig. 1a captures the property that no two philosophers can be talking at the same time. The philosopher currently talking is recorded in variable x, which can only be rebound after that philosopher stops talking. If a different philosopher starts talking before this happens this is an error. Consider the trace $\tau_1 = \texttt{start}(1).\texttt{stop}(1).\texttt{start}(2)$. This is in $\mathcal{L}(\mathsf{Phil})$ as the run $\langle 1, [\,] \rangle \xrightarrow{\texttt{start}(1)} \langle 2, [x \mapsto 1] \rangle \xrightarrow{\texttt{stop}(1)} \langle 1, [x \mapsto 1, y \mapsto 1] \rangle \xrightarrow{\texttt{start}(2)} \langle 2, [x \mapsto 2, y \mapsto 1] \rangle$ ends in a final state. However, the trace $\tau_2 = \texttt{start}(1).\texttt{start}(2)$ is not in $\mathcal{L}(\mathsf{Phil})$ as the run $\langle 1, [\,] \rangle \xrightarrow{\texttt{start}(1)} \langle 2, [x \mapsto 1] \rangle \xrightarrow{\texttt{start}(2)} \langle 3, [x \mapsto 1, y \mapsto 2] \rangle$ does not end in a final state.

Auction Bidding. The EA Hat in Fig. 1b captures the property that bids on item 'hat' must be strictly increasing. Consider the trace $\tau_3 = \texttt{bid}(\text{'hat'}, 1).\texttt{bid}(\text{'hat'}, 10).\texttt{bid}(\text{'hat'}, 5)$. The only run on τ_3 is $\langle 1, [\,] \rangle \xrightarrow{\texttt{bid}(\text{'hat'},1)} \langle 2, [max \mapsto 1] \rangle \xrightarrow{\texttt{bid}(\text{'hat'},10)} \langle 2, [max \mapsto 10, new \mapsto 10] \rangle \xrightarrow{\texttt{bid}(\text{'hat'},5)} \langle 3, [max \mapsto 10, new \mapsto 5] \rangle$. As state 3 is non-final, $\tau_3 \notin \mathcal{L}(\mathsf{Hat})$. In this example, a guard is used to capture the failing behaviour out of state 2 and an assignment is used to keep track of the maximum bid.

Candidate Selection. The EA Candi in Fig. 1c captures the property that voter tom is a member of the red party, candidate flo is a candidate for the red party and voter tom ranks flo in position r - a variable. State 2 is accepting as tom only needs to rank flo if she is a candidate for the red party. The more general case is dealt with in the next section by replacing values 'tom', 'flo', and 'red' by quantified variables.

4 Quantified Event Automata

We now define Quantified Event Automata (QEA), which generalise EA by quantifying over zero or more of the variables used in an EA. Acceptance is decided by replacing these quantified variables by each value in their domain to generate a set of EA and then

using the quantifiers to determine which of these EA must accept the given trace. We begin by considering the syntax of QEA and then present their acceptance condition, finishing by returning to our three running examples.

4.1 Syntax

A QEA consists of an EA with some (or none) of its variables quantified by $\forall$ or $\exists$. The domain of each quantified variable is derived from the trace. The variables of an EA are those that appear in its alphabet:

$$\texttt{vars}(\mathsf{E}) = \{x \mid \exists e(\overline{s}) \in \mathsf{E}.\mathcal{A} : x \in \overline{s} \ \wedge \ x \in \mathit{Var}\}.$$

Not all variables need to be quantified. Unquantified variables are left free in E and can be rebound during the processing of the trace - as seen in the previous section.

Definition 8 (Quantified Event Automaton). *A QEA is a pair $\langle \Lambda, \mathsf{E} \rangle$ where E is an EA and $\Lambda \in (\{\forall, \exists\} \times \texttt{vars}(\mathsf{E}) \times \mathit{Guard})^*$ is a list of quantified variables with guards.*

A QEA is well-formed if Λ contains each variable in $\texttt{vars}(\mathsf{E})$ at most once. In the following, we consider a QEA $\mathsf{Q} = \langle \Lambda, \mathsf{E} \rangle$.

4.2 Acceptance

In Sec. 3 we defined the language of an EA. The intuitive idea here is to use the EA E in QEA Q as a template for generating a set of EA and then check if the trace is in the language of each generated EA. To do this, quantified variables in E are replaced by the values taken from the domain of these quantified variables. First we introduce the concept of EA instantiation to replace variables in E with values.

Definition 9 (Event Automaton Instantiation). *Given a binding θ, let $\mathsf{E}(\theta) = \langle Q, \mathcal{A}(\theta), \delta(\theta), q_0, F \rangle$ be the θ-instantiation of E where*

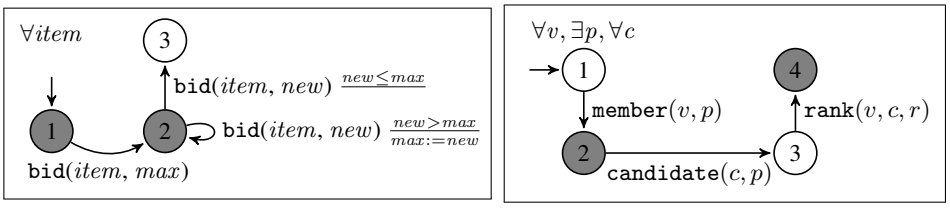
$$\begin{aligned} \mathcal{A}(\theta) \ &= \ \{\mathbf{b}(\theta) \mid \mathbf{b} \in \mathcal{A}\} \\ (q, \mathbf{b}(\theta), g', \gamma', q') \in \delta(\theta) \ \textit{iff} \ &(q, \mathbf{b}, g, \gamma, q') \in \delta \ \textit{and} \ g'(\varphi) = g(\theta \dagger \varphi) \\ &\textit{and} \ \gamma'(\varphi) = \gamma(\theta \dagger \varphi). \end{aligned}$$

The domain of each quantified variable is derived from the values in the trace. The intuition here is that the events $\texttt{start}(x)$ and $\texttt{stop}(x)$ allow us to identify the values that the quantified variable x can take. Therefore, the domain for x is computed by finding all values bound to x when matching any event in the trace with any event in the alphabet of the EA that uses x.

Definition 10 (Derived Domain). *The derived domain of a trace τ is a map from variables quantified in Λ to sets of values:*

$$\mathit{Dom}(\tau)(x) = \{\texttt{match}(\mathbf{a}, \mathbf{b})(x) \mid \mathbf{b} = e(..., x, ...) \in \mathcal{A} \wedge \mathbf{a} \in \tau \wedge \texttt{matches}(\mathbf{a}, \mathbf{b})\}.$$

Each instantiation of E is concerned only with the behaviour of a small set of values (or the events using those values) but a trace can contain other values - that is for binding θ a trace can contain events not in $\texttt{ground}(\mathsf{E}(\theta))$. We need to restrict the trace so that we can test whether it is in the language of $\mathsf{E}(\theta)$. We do this by filtering out any event not in $\texttt{ground}(\mathsf{E}(\theta))$. Note that our notion of projection is w.r.t. a set of parametric events (captured by an EA), which differs from the projection in parametric trace slicing (Definition 1) done w.r.t. a binding. Therefore, we are able to deal with event names which are associated with multiple different variable lists.



(a) Auction Bidding QEA (b) Candidate Selection QEA

Fig. 2: Two QEAs.

Definition 11 (Projection). *The projection of* $\tau \in \mathit{Trace}$ *w.r.t.* E *is defined as:*

$$\epsilon \downarrow_{\mathsf{E}} = \epsilon \qquad \mathbf{a}.\tau \downarrow_{\mathsf{E}} = \begin{cases} \mathbf{a}.(\tau \downarrow_{\mathsf{E}}) & \textit{if } \mathbf{a} \in \mathtt{ground}(\mathsf{E}), \\ (\tau \downarrow_{\mathsf{E}}) & \textit{otherwise}. \end{cases}$$

A trace τ satisfies the property w.r.t. a binding θ iff $\tau \downarrow_{\mathsf{E}(\theta)} \in \mathcal{L}(\mathsf{E}(\theta))$. Note that we could use a different formalism to define such a language, or alter the semantics of EA, and this notion of satisfaction would remain unchanged. Finally, the quantifiers use the derived domain to inductively generate bindings and dictates which of these bindings the trace must satisfy the property with respect to.

Definition 12 (Acceptance). Q *accepts a ground trace* τ *if* $\tau \models_{[\,]} \Lambda.\mathsf{E}$ *where* $\models_\theta$ *is defined as*

$\tau \models_\theta (\forall x : g)\Lambda'.\mathsf{E}$ *iff for all* d *in* $Dom(\tau)(x)$ *if* $g(\theta \dagger [x \mapsto d])$ *then* $\tau \models_{\theta\dagger[x\mapsto d]} \Lambda'.\mathsf{E}$
$\tau \models_\theta (\exists x : g)\Lambda'.\mathsf{E}$ *iff for some* d *in* $Dom(\tau)(x)$ $g(\theta \dagger [x \mapsto d])$ *and* $\tau \models_{\theta\dagger[x\mapsto d]} \Lambda'.\mathsf{E}$
$\tau \models_\theta \epsilon.\mathsf{E}$ *iff* $\tau \downarrow_{\mathsf{E}(\theta)} \in \mathcal{L}(\mathsf{E}(\theta))$

Universal quantification (resp. existential) means that a trace must satisfy the property w.r.t. all (resp. at least one) generated bindings.

4.3 Examples

We revisit the examples introduced in Sec. 2 and used in Sec. 3.

Talking Philosophers The EA in Fig. 1a can be treated directly as a QEA with no quantifications - in this case a single global value (x) is used to record the currently talking philosopher and we are not concerned with the behaviour of individual philosophers in isolation.

Auction Bidding The QEA in Fig. 2a captures the general Auction Bidding property. The quantifications indicate that only the $item$ variable should be instantiated, thus leaving the max and new variables free to be rebound whilst processing the trace.

Candidate Selection The QEA Select in Fig 2b captures the general Candidate Selection property that for every voter there is a party that the voter is a member of, and the voter ranks all candidates for that party. Let us consider the following trace τ_4

`member`('tom','red').`member`('ali','blue').`candidate`('jim','red').`candidate`('flo','red').
`candidate`('don','blue').`rank`('tom','jim',1).`rank`('ali','don',1).

In this trace ali ranks all candidates for the blue party but tom only ranks one of the candidates for the red party. The derived domain is $Dom(\tau_4) = [v \mapsto \{\text{'tom','ali'}\}, p \mapsto \{$

'red', 'blue' $\}, c \mapsto \{$'jim', 'flo', 'don'$\}]$ leading to 12 possible bindings. For space reasons we do not enumerate these bindings here, but leave it to the reader to verify that for five of these bindings the instantiated EA accepts the trace, and that adding the event `rank`('tom','flo',2) to the trace would make the trace accepting.

5 Step-wise Evaluation of QEA

In the previous section, we presented an acceptance condition to decide whether a trace satisfies the property represented by a QEA. This first built up the derived domain by inspecting the trace, then used bindings generated from this domain to generate a set of instantiated EA, checked whether the trace was in the language of each instantiated EA and finally used this information, along with quantifiers, to decide whether the trace was accepted. This requires us to pass over the trace at least twice - first to generate the bindings and then to check the EAs instantiated with them.

For runtime verification purposes, we need to combine these two passes into one – passing over the trace as it is produced. To do this we process each event when it arrives by building the derived domain and keeping track of the status of each instantiated EA on the fly. To decide the acceptance of the trace received up to a certain point we need to compute the information required by Def. 12. This can be split into two concerns

1. *Building the Derived Domain.* When a new event is received the values it contains must be recorded. These values are obtained by matching (as per Def. 4) the received event with the events in the alphabet of the given EA.
2. *Tracking the status of Instantiated EAs.* The relevant bindings for Def. 12 are those that can be generated from the derived domain and that bind all quantified variables - we call such bindings *total*. We need to track the status of the EA instantiated with each such total binding.

For efficiency reasons, we capture the derived domain in the bindings that can be built from it, instead of storing this separately. The status of each instantiated EA can be captured by the configurations reachable by the trace received so far projected with respect to that instantiated EA. In the following we break this down into three steps:

1. Generating bindings and associating with them the relevant projected traces
2. Adapting this approach to generate configurations rather than projected traces
3. Showing how acceptance can be decided based on these configurations

When considering runtime verification, efficiency is obviously a major concern. We do not present an optimised algorithm here, but keep optimisation in mind when discussing design decisions. The approach presented here can be optimised in a number of ways - note that the main structure of the approach is similar to that taken by JAVAMOP, and therefore many optimisations applied in this tool would be applicable here.

We illustrate how to monitor a trace in a step-wise fashion by discussing the Candidate Selection example. We consider how the data structures relevant for monitoring are built up for the QEA Select in Fig 2b and the trace τ_4 given on page 8.

5.1 Generating Projections

In this section we show how to use a trace and a QEA to construct a *monitoring state*, which associates bindings with projected traces:

$$MonitoringState = Binding \rightharpoonup Trace$$

Partial bindings		Total bindings	
[ ]	$\mapsto \epsilon$	$[v \mapsto$ t, $p \mapsto$ r, $c \mapsto$ j$]$	$\mapsto$ mem(t,r).can(j,r).ran(t,j,1)
$[v \mapsto$ t, $p \mapsto$ r$]$	$\mapsto$ mem(t,r)	$[v \mapsto$ t, $p \mapsto$ r, $c \mapsto$ f$]$	$\mapsto$ mem(t,r).can(f,r)
$[v \mapsto$ a, $p \mapsto$ b$]$	$\mapsto$ mem(a,r)	$[v \mapsto$ t, $p \mapsto$ r, $c \mapsto$ d$]$	$\mapsto$ mem(t,r)
$[p \mapsto$ r, $c \mapsto$ j$]$	$\mapsto$ can(j,r)	$[v \mapsto$ t, $p \mapsto$ b, $c \mapsto$ d$]$	$\mapsto$ can(d,b)
$[p \mapsto$ r, $c \mapsto$ f$]$	$\mapsto$ can(f,r)	$[v \mapsto$ a, $p \mapsto$ r, $c \mapsto$ j$]$	$\mapsto$ can(j,r)
$[p \mapsto$ b, $c \mapsto$ d$]$	$\mapsto$ can(d,b)	$[v \mapsto$ a, $p \mapsto$ r, $c \mapsto$ f$]$	$\mapsto$ can(f,r)
		$[v \mapsto$ a, $p \mapsto$ b, $c \mapsto$ j$]$	$\mapsto$ mem(a,b)
		$[v \mapsto$ a, $p \mapsto$ b, $c \mapsto$ f$]$	$\mapsto$ mem(a,b)
		$[v \mapsto$ a, $p \mapsto$ b, $c \mapsto$ d$]$	$\mapsto$ mem(a,b).can(d,b).ran(a,d,1)

Table 1: The monitor state generated by monitoring τ_4 for Select. Event names have been truncated to three letters and parameter values to their first letter.

The monitoring state for our example is given in Table 1. Note that the bindings contain quantified variables only. Let us consider how this monitoring state was built starting with the empty monitoring state $[\,] \mapsto \epsilon$. We first examine the QEA Select and note that its alphabet is $\{\texttt{member}(v, p), \texttt{candidate}(c, p), \texttt{rank}(v, c, r)\}$.

On observing τ_4's first event, member('tom','red'), we construct the binding $[v \mapsto$ 'tom', $p \mapsto$ 'red'$]$ by matching with member(v,p). This binding is added to the monitoring state, along with the associated projected trace. We process τ_4's second event in a similar way to add:

$$[v \mapsto \text{'tom'}, p \mapsto \text{'red'}] \mapsto \texttt{member}(\text{'tom'}, \text{'red'})$$
$$[v \mapsto \text{'ali'}, p \mapsto \text{'blue'}] \mapsto \texttt{member}(\text{'ali'}, \text{'blue'})$$

We did not add single bindings such as $[v \mapsto$ 'tom'$]$ as the projected traces associated with these bindings would be empty, and therefore recording them would be redundant. We only record bindings for which the projected trace is non-empty. On observing τ_4's third event, candidate('jim','red'), we construct the binding $[p \mapsto$ 'red', $c \mapsto$ 'jim'$]$ by matching with candidate(c,p) and add this to the monitoring state:

$$[p \mapsto \text{'red'}, c \mapsto \text{'jim'}] \mapsto \texttt{candidate}(\text{'jim'}, \text{'red'})$$

We then combine this binding with the existing binding $[v \mapsto$ 'tom', $p \mapsto$ 'red'$]$ to get the binding $[v \mapsto$ 'tom', $c \mapsto$ 'jim', $p \mapsto$ 'red'$]$. The projected trace for this new binding is the trace associated with the original binding extended with the current event.

$$[v \mapsto \text{'tom'}, p \mapsto \text{'red'}, c \mapsto \text{'jim'}] \mapsto \texttt{member}(\text{'tom'}, \text{'red'}).\texttt{candidate}(\text{'jim'}, \text{'red'})$$

To see why we do this recall that the submap relation $\sqsubseteq$ gives a partial order on bindings – illustrated in Fig.3. By definition, the projected trace for a new binding will include all the projected traces for existing bindings it subsumes w.r.t. $\sqsubseteq$, and when it is created all such events are captured by the *largest* such existing binding. Like JAVAMOP we call this notion *maximality*. Therefore, the projected trace mapped to by a new binding must extend the projected trace mapped to by the maximal existing binding.

As noted earlier, we must record any binding that can be built from the derived domain that has a non-empty projection. We can build two such bindings by combining submaps of $[v \mapsto$ 'tom', $p \mapsto$ 'red'$]$ with existing bindings as follows:

$$[v \mapsto \text{'ali'}, p \mapsto \text{'red'}, c \mapsto \text{'jim'}] \mapsto \texttt{candidate}(\text{'jim'}, \text{'red'})$$
$$[v \mapsto \text{'ali'}, p \mapsto \text{'blue'}, c \mapsto \text{'jim'}] \mapsto \texttt{member}(\text{'ali'}, \text{'blue'})$$

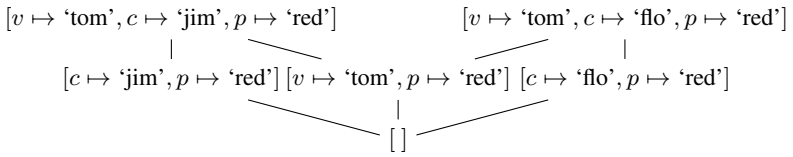


Fig. 3: A subset of bindings from Table 1 ordered by the submap relation.

We now formalise how these bindings and projected traces are generated. We will need to select the quantified part of a binding, hence, for an assumed quantifier list Λ, let $\texttt{quantified}(\theta) = [(x \mapsto v) \in \theta \mid x \in \texttt{vars}(\Lambda)]$ where $\texttt{vars}(\Lambda) = \{x \in Var \mid (\_, x, \_) \in \Lambda\}$. A binding θ is *total* if $\underline{dom}(\theta) = \texttt{vars}(\Lambda)$ and *partial* otherwise.

An event is added to the projection for a binding θ if it matches with an event in $\mathsf{E}(\theta).\mathcal{A}$ and the resulting binding does not contain quantified variables - this second part is necessary as θ may be partial. For example, `member`('tom','red') is not added to the projection for $[\,]$ as the binding that makes it match with $\texttt{member}(c, p)$ binds c and p.

Definition 13 (Event Relevance). *A ground event a is* relevant *to a binding θ iff*

$$\exists \mathbf{b} \in \mathcal{A}(\theta) : \texttt{matches}(\mathbf{a}, \mathbf{b}) \wedge \texttt{quantified}(\texttt{match}(\mathbf{a}, \mathbf{b})) = [\,]$$

To extend a binding θ we first find all bindings that match the received event with an event in $\mathsf{E}(\theta).\mathcal{A}$, and then compute all possible extensions to θ based on these bindings. If the received event is relevant to a generated new binding we add this event to the previous trace, otherwise the previous trace is just copied.

Definition 14 (Extending a Binding). *Let $direct(\theta, \mathbf{a})$ be the bindings that directly extend θ given $\mathbf{a}$, defined in terms of those bindings that can be built from $\mathbf{a}$.*

$$from(\theta, \mathbf{a}) = \{\texttt{quantified}(\texttt{match}(\mathbf{a}, \mathbf{b})) \mid \mathbf{b} \in \mathcal{A}(\theta)\}$$
$$direct(\theta, \mathbf{a}) = \{\theta \dagger \theta' \mid \exists \theta'' \in from(\theta, \mathbf{a}) : \theta' \sqsubseteq \theta'' \wedge\ \theta' \neq [\,]\}$$

Let $all(\theta, \mathbf{a})$ be the smallest superset of $direct(\theta, \mathbf{a})$ containing $\theta_1 \dagger \theta_2$ for all compatible θ_1 and θ_2 in $direct(\theta, \mathbf{a})$. The required extensions $\texttt{extend}(\mathbf{a}, \theta, \sigma)$ are given by

$$(\theta' \mapsto \sigma') \in \texttt{extend}(\mathbf{a}, \theta, \sigma) \text{ iff } \theta' \in all(\theta, \mathbf{a}) \wedge \sigma' = \begin{cases} \sigma.\mathbf{a} & \text{if } \exists \theta'' \in from(\theta, \mathbf{a}) : \theta'' \sqsubseteq \theta' \\ \sigma & \text{otherwise} \end{cases}$$

On receiving a new event the next monitoring state is built by iterating through the current monitoring state and, for each binding, adding the event to its associated projected trace if it is relevant and adding new extending bindings as described above. To ensure that new bindings extend the maximal existing binding we iterate through bindings in the reverse order defined by $\sqsubseteq$ and only add bindings that do not already exist. This ensures that the maximal existing binding for a new binding will always be encountered first. By adding all bindings that extend existing bindings we ensure that the derived domain is correctly recorded and that all necessary total bindings will be created.

Definition 15 (Single Step Monitoring Construction). *Given ground event $\mathbf{a}$ and monitoring state M. Let $\theta_1, \ldots, \theta_m$ be a linearisation of the domain of M i.e. if $\theta_j \sqsubset \theta_k$ then $j > k$ and every element in the domain of M is present once in the sequence, hence $m = |M|$. We define $(\mathbf{a} * M) = N_m \in MonitoringState$ where N_m is iteratively defined as follows for $i \in [1, m]$.*

$$N_0 = [\,] \qquad N_i = N_{i-1} \dagger \texttt{Add}_i \dagger \begin{cases} [\theta_i \mapsto M(\theta_i).\mathbf{a}] & \text{if } \mathbf{a} \text{ is relevant to } \theta_i \\ [\theta_i \mapsto M(\theta_i)] & \text{otherwise} \end{cases}$$

where $\texttt{Add}_i = [(\theta' \mapsto \sigma') \in \texttt{extend}(\mathbf{a}, \theta_i, M(\theta_i)) \mid \theta' \notin \underline{dom}(N_{i-1})]$

$[\,] \mapsto \langle 1, [\,]\rangle$	$[p \mapsto \text{b}, c \mapsto \text{d}] \mapsto \langle 1, [\,]\rangle$	$[v \mapsto \text{t}, p \mapsto \text{b}, c \mapsto \text{d}] \mapsto \langle 1, [\,]\rangle$
$[p \mapsto \text{r}, c \mapsto \text{j}] \mapsto \langle 1, [\,]\rangle$	$[v \mapsto \text{a}, p \mapsto \text{b}, c \mapsto \text{d}] \mapsto \langle 4, [r \mapsto 1]\rangle$	$[v \mapsto \text{a}, p \mapsto \text{r}, c \mapsto \text{j}] \mapsto \langle 1, [\,]\rangle$
$[v \mapsto \text{t}, p \mapsto \text{r}] \mapsto \langle 2, [\,]\rangle$	$[v \mapsto \text{t}, p \mapsto \text{r}, c \mapsto \text{j}] \mapsto \langle 4, [r \mapsto 1]\rangle$	$[v \mapsto \text{a}, p \mapsto \text{r}, c \mapsto \text{f}] \mapsto \langle 1, [\,]\rangle$
$[p \mapsto \text{r}, c \mapsto \text{f}] \mapsto \langle 1, [\,]\rangle$	$[v \mapsto \text{t}, p \mapsto \text{r}, c \mapsto \text{f}] \mapsto \langle 3, [\,]\rangle$	$[v \mapsto \text{a}, p \mapsto \text{b}, c \mapsto \text{j}] \mapsto \langle 2, [\,]\rangle$
$[v \mapsto \text{a}, p \mapsto \text{b}] \mapsto \langle 2, [\,]\rangle$	$[v \mapsto \text{t}, p \mapsto \text{r}, c \mapsto \text{d}] \mapsto \langle 2, [\,]\rangle$	$[v \mapsto \text{a}, p \mapsto \text{b}, c \mapsto \text{f}] \mapsto \langle 2, [\,]\rangle$

Table 2: The monitor lookup generated by monitoring τ_4 for Select. Parameter values have been truncated to their first letter.

Finally, for the input trace the construction of Def. 15 is applied to each event, starting with an initial monitoring state - the empty binding with the empty projection.

Definition 16 (Stepwise Monitoring). *For a trace $\tau = a_0.a_1 \ldots a_n$ we define the final monitoring state M_τ as $a_n * (\ldots * (a_0 * [\,[\,] \mapsto \epsilon\,]) \ldots)$.*

The final monitoring state M_τ contains the information required to decide whether the trace τ is accepted (as specified in Def. 12) (discussed in Sec. 5.3).

5.2 Generating Configurations

In the previous section we associated bindings with projected traces. However, using projected traces directly would not be efficient, especially as we would need to run through each projected trace to decide the status of acceptance on each step. Instead, we record the configurations reachable by those projections. To do this we define a new structure:

$$MonitorLookup = Binding \rightharpoonup \mathcal{P}(Config)$$

The monitor lookup for our example is given in Table 2. We adapt the construction in the previous section to build a monitor lookup by defining a function in Algorithm 1 that computes the next configurations for a binding given a received event. Let $\texttt{unquantified}(\theta) = \theta \backslash \texttt{quantified}(\theta)$. We can modify Def. 15 to produce a monitoring lookup instead of a monitoring state by replacing the inductive definition of N_i with

$$N_i = N_{i-1} \dagger \texttt{Add}_i \dagger [\theta_i \mapsto \textsc{Next}(\theta_i, \mathbf{a}, M(\theta_i))]$$

where $\texttt{Add}_i = [\theta \mapsto \textsc{Next}(\theta, \mathbf{a}, M(\theta_i)) \mid \theta \in all(\theta_i, \mathbf{a})]$. This processes the projected trace for a binding as it is produced as NEXT(θ,**a**,C) gives all configurations reachable by event **a** from configurations in C on $\mathsf{E}(\theta)$, staying in the same configuration if no transition can be taken. Because of this last point no changes are made if the event is not relevant to the binding. The check that $\texttt{quantified}(\varphi') = [\,]$ ensures that no new quantified variables are bound when taking a transition. Note that this function relies on the wait semantics of EA and could not necessarily be used without modification if we were to replace EA with an alternative formalism - the previous construction only assumes an alphabet of events to construct projected traces.

Algorithm 1 Finding the next configurations when adding an event to a projection.

```
function NEXT(θ : Binding, a : GEvent,
              C : Set[Config]) :Set[Config]
    next ← ∅
    for ⟨q, φ⟩ in C do
        for (q1, b, g, γ, q2) ∈ E.δ do
            if q1=q ∧ matches(a, b(θ)) then
                φ' ← φ † match(a, b(θ))
                if quantified(φ')=[ ] and
                   g(unquantified(φ')) then
                    next ← next + ⟨q2,
                        γ(unquantified(φ'))⟩
        if no transitions are taken then
            next ← next ∪⟨q, φ⟩
    return next
```

5.3 Acceptance

Here we consider when a monitor lookup is accepted. We adapt the notion of acceptance given in Def. 12 to detect success or failure as soon as it is possible. We define a four valued verdict domain containing the classifications *Strong Success*, *Weak Success*, *Strong Failure* and *Weak Failure*. The strong versions of success and failure indicate that no extensions of the trace can alter the verdict. For example, τ_3 is strongly failing for the QEA in Fig. 2a as no extensions will be accepted. We first identify the special states of E such that all extensions of trace τ reaching that state will be in $\mathcal{L}(\mathsf{E})$ iff τ is.

Definition 17 (Strong Success and Failure States). *Let* $\texttt{reach}(q)$ *be the set of reachable states of* $q \in Q$*. Let* $\texttt{strongS} = \{q \in F \mid \texttt{reach}(q) \subseteq F\}$ *be the strong success states. Let* $\texttt{strongF} = \{q \in Q \backslash F \mid \texttt{reach}(q) \cap F = \emptyset\}$ *be the strong failure states. Note that it is not necessarily the case that* $(\texttt{strongS} \cup \texttt{strongF}) = Q$*.*

If all quantifiers are universal then all total bindings must reach successful configurations, and if one cannot (i.e., is in a strongly failing state) a strong failure can be reported, similarly if all quantifiers are existential then a single total binding reaching a configuration in a strongly successful state means that strong success can be reported. Strong success or failure cannot be reported where we have a mix of existential and universal quantification.

Definition 18 (Monitor Lookup Classification). *We define the function* $\texttt{Check}(L, \mathsf{Q})$ *to decide whether a monitor lookup* L *satisfies a QEA* Q*. Let* `uni` *be true if all quantifiers in* $\mathsf{Q}.\Lambda$ *are universal,* `exi` *be true if they are all existential and* G *be the combination of all guards in* $\mathsf{Q}.\Lambda$*, i.e,* $G(\theta)$ *iff* $\forall(\_, \_, g) \in \Lambda : g(\theta)$*.*

$$\texttt{Check}(L, \mathsf{Q}) = \begin{cases} \textit{StrongSuccess iff } \texttt{exi} \wedge \exists \theta \in \underline{dom}(L) : G(\theta) \\ \qquad\qquad\qquad\qquad \wedge \exists \langle q, \varphi \rangle \in L(\theta) : q \in \texttt{StrongS} \\ \textit{StrongFailure iff } \texttt{uni} \wedge \exists \theta \in \underline{dom}(L) : G(\theta) \\ \qquad\qquad\qquad\qquad \wedge \forall \langle q, \varphi \rangle \in L(\theta) : q \in \texttt{StrongF} \\ \textit{WeakSuccess iff not a strong result and } L \models_{[\,]} \mathsf{Q}.\Lambda \\ \textit{WeakFailure iff not a strong result and } L \not\models_{[\,]} \mathsf{Q}.\Lambda \end{cases}$$

for $L \models_\theta \Lambda$*, defined as*

$$\begin{array}{ll} L \models_\theta (\forall x : g)\Lambda' & \textit{iff for all } d \textit{ in } D_L(x) \textit{ if } g(\theta \dagger [x \mapsto d]) \textit{ then } L \models_{\theta \dagger [x \mapsto d]} \Lambda' \\ L \models_\theta (\exists x : g)\Lambda' & \textit{iff for some } d \textit{ in } D_L(x)\ g(\theta \dagger [x \mapsto d]) \textit{ and } L \models_{\theta \dagger [x \mapsto d]} \Lambda' \\ L \models_\theta \epsilon & \textit{iff } \begin{cases} \exists \langle q, \varphi \rangle \in L(\theta) : q \in E.F & \textit{if } \theta \in \underline{dom}(L) \\ q_0 \in F & \textit{otherwise} \end{cases} \end{array}$$

where $D_L(x) = \{\theta(x) \mid \theta \in \underline{dom}(L) \wedge x \in \underline{dom}(\theta)\}$

Note that if $\theta \notin \underline{dom}(L)$ then there were no events relevant to θ in the trace and therefore the projected trace for θ is empty. An efficient algorithm for computing $\texttt{Check}(L, \mathsf{Q})$ would keep track of the current status and update this whenever a relevant change is made to the monitor lookup, rather than recomputing it on each step.

The monitor lookup in Table. 2 is weakly failing. The monitor lookup for the trace $\tau_4.\texttt{rank}(\text{'tom'}, \text{'flo'}, 2)$ is weakly successful as this changes the configuration associated with $[c \mapsto \mathrm{t}, p \mapsto \mathrm{r}, c \mapsto f]$ to $\langle 4, [r \mapsto 2] \rangle$ and then tom ranks all candidates for the red party and ali ranks all candidates for the blue party. Observe that it is important that state 2 is accepting – as voters only need to rank candidates for the given party.

6 Related Work

QEA extends the parametric trace slicing approach [7] taken by JAVAMOP [10] by allowing event names to be associated with multiple different variable lists, by allowing non-quantified variables to vary during monitoring, and by allowing existential quantification in addition to universal quantification. This results in a strictly more expressive logic. JAVAMOP can be considered as a framework supporting parameterization for any propositional logic, provided as a plugin. QEA is similarly composed of quantification added to event automata, which can be replaced with other forms of logic. Parametric trace slicing can be seen as a special case of our notion of projection used to define whether a trace is in the language of a monitor for some binding.

TRACEMATCHES [1] is an extension of `AspectJ` where specifications are given as regular expressions over pointcuts. Parametric properties are monitored rather efficiently, but TRACEMATCHES, like JAVAMOP, suffers from the the limitation that each event name is associated with a unique list of variables.

A number of expressive techniques supporting data parameterization are based on rewriting. EAGLE [2] is based on rewriting of temporal logic formulas. For each new event, a formula is rewritten into a new formula that has to hold in the next step. RULER [2, 5] supports a specification language based on explicit rewrite-rules. Parameterized state machines are supported by LOGSCOPE [3] and TRACECONTRACT [4]. TRACECONTRACT is defined as an internal DSL in `Scala` (an API), re-using `Scala`'s language constructs, including for example pattern matching. In both cases, states are explicitly parameterized with data, similar to how for example functions in a programming language are parameterized. A variant of LOGSCOPE has been created (not described in [3]) where the notion of maximality can be encoded by allowing transitions to refer to the presence (or non-presence) of other states with specific bindings as guards.

JLO [13] is a parameterized LTL, from which monitors are generated. A formula is rewritten into a new formula for each new event, as in EAGLE. JLO events are defined by pointcuts inspired by aspect-oriented programming, and monitors are generated as `AspectJ` aspects. An embedding of LTL in `Haskell` is described in [14]. It is similar to TRACECONTRACT, but whereas TRACECONTRACT handles data parameterization by re-using `Scala`'s built-in notion of partial functions and pattern matching, [14] introduces a concept called *formula templates* instantiated for all possible permutations of propositions. Stolz introduced temporal assertions with parametrized propositions [12] with a similar aim of adding free variables and quantification to a runtime monitoring formalism (next-free LTL). The main distinction wrt. this work is the treatment of quantification - in [12] the domain of quantification is based on the current state only.

In Sec. 5 we use a four-valued verdict domain, which have been explored previously in the context of runtime monitoring, for example in [6], also RULER uses a four-valued logic.

7 Conclusion and Future Work

We have introduced a new formalism for parametric runtime monitoring that is more expressive than the current most efficient techniques. We have presented both big-step and small-step semantics for our new formalism. Although not described in this pa-

per, we have used these small-step semantics to implement a basic runtime monitoring algorithm in `Scala` and carried out initial testing.

We plan to explore four main areas of future work. Firstly, we intend to explore further the language theoretic properties of QEA. Secondly, we wish to explore different efficient runtime monitoring implementations. As our approach generalises the parametric trace slicing approach we may adapt optimisations implemented in JAVAMOP. Our stepwise construction also allows for alternative optimisations. Thirdly, we wish to consider the utility of QEA as a specification language. So far we have structured the development to separate EA, which define properties of specific sets of values, and QEA which generalise these. We may exploit this separation to replace EA with different formalisms, such as regular or context-free grammars, leading to a more general framework for specifying properties. These replacements would be to increase the usability rather than expressiveness of the framework. Finally, whilst developed in the context of runtime verification, the ideas in this paper appear to be relevant to specification mining (attempting to derive specifications by examining patterns in traces) and we intend to explore this link further.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, October 2005.
2. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.
3. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 2010.
4. H. Barringer and K. Havelund. Tracecontract: a Scala DSL for trace analysis. In *Proc. of the 17th international conference on Formal methods*, pages 57–72, Berlin, Heidelberg, 2011.
5. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RuleR. *J Logic Computation*, 20(3):675–706, June 2010.
6. A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of the 7th international conference on Runtime verification*, RV'07, pages 126–138, Berlin, Heidelberg, 2007. Springer-Verlag.
7. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *TACAS '09*, pages 246–261, Berlin, Heidelberg, 2009.
8. K. Havelund and A. Goldberg. Verify your runs. In *Verified Software: Theories, Tools, Experiments, VSTTE 2005*, pages 374–383, Berlin, Heidelberg, 2008.
9. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2008.
10. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the mop runtime verification framework. *J Software Tools for Technology Transfer*, pages 1–41, 2011.
11. Runtime Verification. http://www.runtime-verification.org, 2001-2011.
12. V. Stolz. Temporal assertions with parametrized propositions*. *J. Log. and Comput.*, 20:743–757, June 2010.
13. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 109–124. Elsevier, 2006.
14. V. Stolz and F. Huch. Runtime verification of concurrent Haskell programs. In *Proc. of the 4th Int. Workshop on Runtime Verification (RV'04)*, volume 113 of *ENTCS*, pages 201–216. Elsevier, 2005.