From Parametric Trace Slicing to Rule Systems

Giles Reger and David Rydeheard

September 23, 2018

From Parametric Trace Slicing to Rule Systems

Giles Reger* and David Rydeheard

University of Manchester, Manchester, UK

Abstract. Parametric runtime verification is the process of verifying properties of execution traces of (data carrying) events produced by a running system. This paper continues our work exploring the relationship between specification techniques for parametric runtime verification. Here we consider the correspondence between trace-slicing automata-based approaches and rule-systems. The main contribution is a translation from quantified automata to rule-systems, which has been implemented in SCALA. This then allows us to highlight the key differences in how the two formalisms handle data, an important step in our wider effort to understand the correspondence between different specification languages for parametric runtime verification.

1 Introduction

Runtime verification [7, 14, 20] is the process of checking properties of execution traces produced by running a computational system. An execution trace is a finite sequence of *events* generated by the computation. In many applications, events carry *data values* – the so-called parametric, or first-order, case of runtime verification. To apply runtime verification, we need to provide (a) a specification language for describing properties of execution traces, and (b) a mechanism for checking these formally-defined properties during execution, i.e. a procedure for generating monitors from specifications. Many different specification languages for runtime verification have been proposed and almost every new development introduces its own specification language.

This work furthers our broader goal of organising and understanding the space of specification languages for runtime verification. As explained later, we see little reuse of specification languages in runtime verification and little is understood about the relationship between the different languages that have been introduced. We believe that the field can be considerably improved by a better understanding of this space.

This paper specifically explores the relationship between two particular approaches to specification for parametric runtime verification: *parametric trace slicing* and *rule systems*. We begin by describing the general setting we are working in (Section 2) before introducing these two languages (Section 3). The main contribution of the paper is a translation from specifications using parametric trace slicing to those using rules (Section 4). We define the translation, provide some examples, and prove its correctness. The translation has been implemented and validated in SCALA, available online at `https://github.com/selig/qea_to_rules`. A further contribution is then a discussion of the things we have learnt about the relationship between these two languages via the development of the translation (Section 5). We conclude in Section 6.

* The work of this author is supported by COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

2 Setting

In this paper we focus on the runtime verification problem at a level of abstraction where we assume that a run of a system has been abstracted in terms of a finite sequence of events via some instrumentation method. Such techniques are described elsewhere [7].

Defining The Runtime Verification Problem. We begin by defining events, traces, and properties. We assume disjoint sets of event names Σ, variables Var, and values Val. We do not directly consider sorts (e.g. variable x being an integer) as this is not essential to this work, but assume events are well-sorted where it matters.

Definition 1 (Events, Traces, and Properties). *An* event *is a pair of an event name* e *and a list of parameters (variables or values)* $v_1, \ldots, v_n$*, usually written* $\mathtt{e}(v_1, \ldots, v_n)$*. An event is* ground *if it does not contain variables. A* trace *is a finite sequence of ground events. A property is a (possibly infinite) set of traces.*

We use x, y, z for variables and a, b, c or numbers for values (unless context requires otherwise), τ for traces and $\mathcal{P}$ for properties. For example, $\mathtt{login}(x, 42)$ is an event where x is a variable and 42 a value; the finite sequence $\mathtt{login}(a, 42).\mathtt{logout}(a)$ is a trace; and the set $\{\mathtt{login}(a, 42).\mathtt{logout}(a), \mathtt{login}(b, 42).\mathtt{logout}(b)\}$ is a property. We write $\mathbf{a}, \mathbf{b}$ for events where their structure is unimportant.

We say that a property is *propositional* if all events in all traces have empty lists of parameters, otherwise it is *parametric* (or first-order). A *specification language* provides a language for writing specifications φ and provides a semantics that defines the property $\mathcal{P}(\varphi)$ that φ denotes. A specification language is propositional if it can only describe specifications denoting propositional properties, and parametric otherwise.

Definition 2 (The Runtime Verification Problem). *Given a trace* τ *and a specification* φ *decide whether* $\tau \in \mathcal{P}(\varphi)$*.*

Again, we can talk of the *propositional* and *parametric* versions of this problem. The propositional version should be highly familiar - typical specification languages include automata, regular expressions, and linear temporal logic, for which procedures for efficiently deciding the above problem are well known.

There are four main runtime verification approaches that handle the parametric case (see [19] for an overview). Parametric trace slicing [3, 11, 22] separates the issue of quantification from trace-checking using a notion of *projection*. First-order extensions to temporal logic [8, 9, 13, 21, 27] rely on the standard logical treatment of quantification, introducing (somewhat complex) monitor construction techniques to handle this. Rule systems [2, 5, 16] and stream processing [12, 10, 15] do not have inherent notions of quantification. In rule systems values are stored as rule instances (facts) and rules dictate which instances should be added or removed. Stream processing defines sets of stream operators that operate over streams to produce new streams.

We note that there are variations of the above problem e.g. deciding whether $\tau.\tau' \in \mathcal{P}(\varphi)$ for all possible extensions τ' (which acknowledges that finite traces may be prefixes of some infinite trace), or considering a property as a function from traces to some non-boolean verdict domain. In general, the specification languages for such formulations remain the same and much of our work can translate to these variations.

Our Research Question. Given this large space of specification languages our fundamental research question is as follows:

What are the fundamental differences between specification languages for describing parametric properties for runtime verification and how do these differences impact the expressiveness *and* efficiency *of the runtime verification process.*

Below we discuss (i) why we care about this question, and (ii) what our general approach to answering it is.

Why Do We Care? We outline the main motivations behind this research question:

- *Reusable research.* The four main approaches to parametric runtime verification described above have been explored in relative isolation. Developments in one area cannot be easily transferred to another. For example, notions of monitorability and complexity results remain tied to their particular language.
- *Reusable tools, benchmarks, and case studies.* Similarly, tools for one language cannot be directly used for another and related experimental data is tied to that tool. This leads to separate ecosystems where runtime verification solutions are developed in isolation.
- *Balancing Expressiveness and Efficiency.* Some approaches focus on the *expressiveness* of the language before the *efficiency* of the monitoring algorithm, and other approaches have the inverse focus. A key motivation of this work is to see where we can combine the best parts of different approaches. For example, by identifying fragments of an expressive language that can be translated into a language with a more efficient monitoring algorithm.
- *Evaluation.* In general, it is hard to compare approaches without a good understanding of how they are related. The Runtime Verification competition [6, 25] has relied on a manual translation of specifications between languages, which has been problematic in various ways. Ideally, a common language would be used. However, the close links between language and the efficiency of the monitoring algorithm mean that translations would be required from this common language.

Our Approach. We are exploring this broad research question in two complementary directions. Firstly, we are taking an *example-led* approach where we explore concrete examples of specifications in different languages and attempt to infer commonalities, differences, and general relationships. This is ongoing and has begun to highlight conceptual differences between approaches [17–19]. Secondly, we are working towards a general framework for formally exploring the relationship between specification languages. We have chosen to build this via a series of *translations* between approaches. Our previous work [26] introduced a translation from a first-order temporal logic to a language using parametric trace slicing; this current work introduces a translation from parametric trace slicing to rule systems; and we are currently exploring a translation from rule systems to a first-order temporal logic. We believe that these translations can provide a pragmatic way to move between specification languages and highlight the main differences between languages.

3 Two Languages

In this section we introduce two specification languages for parametric runtime verification – one based on parametric trace slicing and the other on rule systems. Examples in both languages are given at the end of the section.

Preliminaries. Let an *event alphabet* $\mathcal{A}(Z)$ be a set of events using variables in Z e.g. $\mathcal{A}(\{x\})$ might be $\{\mathtt{e}(x)\}$ or $\{\mathtt{e}(x), \mathtt{f}(x, x)\}$ but not $\{\mathtt{e}(x), \mathtt{f}(x, y)\}$. A map is a partial function with finite domain. We write $\perp$ for the empty map and $\mathsf{dom}(\theta)$ for the domain of map θ. Given two maps θ_1 and θ_2 we define the following operations:

$$\begin{array}{lll} \mathsf{consistent}(\theta_1, \theta_2) & \mathit{iff} & (\forall x)\; x \in (\mathsf{dom}(\theta_1) \cap \mathsf{dom}(\theta_2)) \rightarrow \theta_1(x) = \theta_2(x) \\ \theta_1 \sqsubseteq \theta_2 & \mathit{iff} & \mathsf{dom}(\theta_1) \subseteq \mathsf{dom}(\theta_2) \;\mathit{and}\; \mathsf{consistent}(\theta_1, \theta_2) \\ (\theta_1 \dagger \theta_2)(x) = v & \mathit{iff} & \theta_2(x) = v \;\mathit{if}\; x \in \mathsf{dom}(\theta_2) \;\mathit{otherwise}\; \theta_1(x) = v \end{array}$$

A valuation is a map from variables to values. We use θ and σ for valuations. Valuations can be applied to structures containing variables to replace those variables.

The sets $\mathit{Guard}(Z)$ and $\mathit{Assign}(Z)$ contain (implicitly well-sorted) guards (boolean expressions) and assignments over the set of variables Z. Such guards denote predicates on valuations with domains in Z, for example $\mathit{Guard}(\{x, y\})$ contains expressions such as $x = y$ and $x \leq 2$. Assignments are finite sequences of the form $x := t$ where $x \in Z$ is a variable and t is an expression over values and variables in Z that can be evaluated with respect to a valuation. We assume a true guard true and an identity assignment id.

Finally, we introduce matching. Given finite parameter sequences $\overline{v}$ and $\overline{w}$, let the predicate $\mathsf{matches}(\overline{v}, \overline{w})$ hold if there is a valuation θ such that $\theta(\overline{v}) = \theta(\overline{w})$. Let $\mathsf{match}(\overline{v}, \overline{w})$ be the minimal such valuation with respect to the sub-map relation $\sqsubseteq$ (if such a valuation exists, undefined otherwise). Let $\mathsf{match}(\overline{v}, \overline{w}, Z)$ be the largest valuation θ such that $\theta \sqsubseteq \mathsf{match}(\overline{v}, \overline{w})$ and $\mathsf{dom}(\theta) \subseteq Z$ i.e. the matching valuation is restricted to Z. We lift all definitions to events by checking equality of event names.

3.1 Parametric Trace Slicing with Quantified Event Automata

Parametric trace slicing [11] was introduced as a technique that transforms a monitoring problem involving quantification *over finite domains* into a propositional one. The idea is to take each valuation of the quantified variables and consider the specification *grounded* with that valuation for the trace *projected* with respect to the valuation. The benefit of this approach is that projection can lead to efficient indexing techniques.

Quantified event automata (QEA) [3] is a slicing-based formalism that generalises previous work on parametric trace slicing. In this work, we consider a restricted form of QEA that does not allow existential quantification (see the discussion in Section 5).

Definition 3 (Quantified Event Automata). *A quantified event automaton is a tuple $\langle X, Q, \mathcal{A}(X \cup Y), \delta, \mathcal{F}, q_0, \sigma_0 \rangle$ where X is a finite set of universally quantified variables, Q is a finite set of states, $\mathcal{A}(X \cup Y)$ is an event alphabet, $\delta \subseteq (Q \times \mathcal{A}(X \cup Y) \times \mathit{Guard}(Y) \times \mathit{Assign}(Y) \times Q)$ is a transition relation, $\mathcal{F} \subseteq Q$ is a set of* final *states, $q_0 \in Q$ is an initial state, and σ_0 is an initial valuation with $\mathsf{dom}(\sigma_0) = Y$.*

The variables Y are implicitly unquantified and are to be used in guards and assignments. An advantage of the parametric trace slicing approach is that the quantified and unquantified parts of the specification can be treated separately. The quantified part is dealt with by trace slicing and the unquantified part is dealt with by the automaton.

Semantics. We now introduce a *small-step* semantics for QEA. We would normally introduce a big-step semantics in terms of the trace slicing operator and use this to motivate the (more operational) small-step presentation. But space does not allow this here and we refer the reader to other texts for this [3, 23]. In the following we assume a fixed QEA of interest and refer to its components e.g. the set of quantified variables X.

Let a *monitoring state* be a map from valuations θ with $\mathsf{dom}(\theta) \subseteq X$ to sets of *configurations*, which are pairs consisting of states $\in Q$ and valuations σ with $\mathsf{dom}(\sigma) = Y$. The small-step semantics defines a construction that extends a monitoring state given a ground event. This construction is then lifted to traces.

Next Configurations. Given a set of configurations P, an event $\mathbf{a}$, and a valuation θ (with $\mathsf{dom}(\theta) = X$), the set $\mathsf{next}(P, \mathbf{a}, \theta)$ of next configurations is defined as the smallest set of configurations such that

$$\left\{ (q', \alpha(\sigma \dagger \mathsf{match}(\mathbf{a}, \mathbf{b}, Y))) \mid \begin{array}{r} \exists (q, \mathbf{b}, \gamma, \alpha, q') \in \delta : \langle q, \sigma \rangle \in P \wedge \mathsf{matches}(\mathbf{a}, \mathbf{b}) \wedge \\ \gamma(\sigma \dagger \mathsf{match}(\mathbf{a}, \mathbf{b}, Y)) \wedge \mathsf{match}(\mathbf{a}, \mathbf{b}, X) \sqsubseteq \theta \end{array} \right\}$$

or P if this set is empty i.e. if no transitions can be taken then P is not updated. This says that we take a transition if we match the event, satisfy the guard, and don't capture any new variables in X not already present in θ.

Relevance. We will update the configurations related to a valuation in the monitoring state if the given event is *relevant* to that valuation. An event $\mathbf{a}$ is *relevant* to some valuation θ if there is an event in the alphabet that matches it consistently with θ i.e.

$$\mathsf{relevant}(\theta, \mathbf{a}) \Leftrightarrow \exists \mathbf{b} \in \mathcal{A}(X \cup Y) : \mathsf{matches}(\mathbf{a}, \mathbf{b}) \wedge \mathsf{match}(\mathbf{a}, \mathbf{b}, X) \sqsubseteq \theta$$

Extensions. We will create a new valuation if matching the given event with an event in the alphabet binds new quantified variables. The set of valuations $\mathsf{extensions}(\theta, \mathbf{a})$ that could extend an existing valuation θ given a new ground event $\mathbf{a}$ can be defined by:

$$\begin{array}{ll} \mathsf{from}(\mathbf{a}) & = \{\theta \mid \exists \mathbf{b} \in \mathcal{A}(X \cup Y) : \mathsf{matches}(\mathbf{a}, \mathbf{b}) \wedge \theta \sqsubseteq \mathtt{match}(\mathbf{a}, \mathbf{b}, X)\} \\ \mathsf{extensions}(\theta, \mathbf{a}) & = \{\theta \dagger \theta' \mid \theta' \in \mathsf{from}(\mathbf{a}) \wedge \mathsf{consistent}(\theta, \theta') \wedge \theta' \neq \bot\} \end{array}$$

This constructs all valuations that can be built directly and then uses the consistent ones.

Construction. We put these together into the monitoring construction.

Definition 4 (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 *from largest to smallest wrt* $\sqsubseteq$ *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 the monitoring state* $(\mathbf{a} * M) = N_m$ *where* N_m *is iteratively defined as follows for* $i \in [1, m]$.

$$N_0 = \bot \qquad N_i = N_{i-1} \dagger \mathtt{Add}_i \dagger \left\{ \begin{array}{ll} [\theta_i \mapsto \mathsf{next}(M(\theta_i), \mathbf{a}, \theta_i)] & \textit{if}\ \mathsf{relevant}(\theta_i, \mathbf{a}) \\ [\theta_i \mapsto M(\theta_i)] & \textit{otherwise} \end{array} \right.$$

where the additions are defined in terms of extensions not already present:

$$\mathtt{Add}_i = [(\theta' \mapsto \mathsf{next}(M(\theta_i), \mathbf{a}, \theta')) \mid \theta' \in \mathsf{extensions}(\theta_i, \mathbf{a}) \wedge \theta' \notin \underline{dom}(N_{i-1})]$$

and next *is a function computing the next configurations given a valuation.*

This construction iterates over valuations (of quantified variables) from *largest* to *smallest* (wrt $\sqsubseteq$). For each valuation it will add any extensions that do not already exist and then update the configuration(s) mapped to by the existing valuation. Let us now consider the aspects that have not yet been defined.

Maximality. The order of traversal in Definition 4 is important as it preserves the principle of maximality. This is the requirement that when we add a new valuation we want to extend the *most informative* or *maximal* valuation as this will be associated with all configurations relevant to the new valuation. Given a set of valuations Θ and a valuation θ let $\mathsf{maximal}(\Theta, \theta) = \theta_M$ be the maximal valuation defined as:

$$\theta_M \in \Theta \wedge \theta_M \sqsubseteq \theta \wedge \forall \theta' \in \Theta : \theta' \sqsubseteq \theta \Rightarrow \theta_M \not\sqsubset \theta'$$

This relies on the fact that $\mathsf{dom}(M)$ is closed under least-upper bounds. In Definition 4, when a valuation θ is introduced its initial set of configurations is taken as those belonging to $\mathsf{maximal}(\mathsf{dom}(M), \theta)$ as otherwise it will already have been added. This principle is important as it makes the later translation complicated.

Quantification Domain. It may not be obvious from the small-step semantics but this semantics ensures that the domain of the monitoring state captures the full cross-product of the quantification domains of X. The domain of variable $x \in X$ is given as

$$\{\mathsf{match}(\mathbf{a}, \mathbf{b})(x) \mid \mathbf{a} \in \tau \wedge \mathbf{b} \in \mathcal{A}(X \cup Y) \wedge \mathsf{matches}(\mathbf{a}, \mathbf{b}) \wedge \mathbf{b} = \mathsf{e}(\ldots, x, \ldots)\}$$

i.e. the set of values in events in the trace that match with events in the alphabet.

The Property Defined by a QEA. Let $M_\tau = \tau * [\perp \mapsto \{(q_0, \sigma_0(Y))\}]$ be the above construction transitively applied to the initial monitoring state. The property defined by the QEA is the set of traces τ such that $\forall \theta \in \mathsf{dom}(M_\tau) : \mathsf{dom}(\theta) = X \Rightarrow \forall (q, \sigma) \in M_\tau(\theta) : q \in F$ i.e. *all* total valuations are only mapped to final states.

3.2 A Rule-Based Approach

We now introduce an approach first introduced in RULER [2] that uses a system of rules to compute a verdict. Our notion of a rule system here could be considered the core of the system introduced in [2] i.e. the extensions in [2] are either trivial or can be defined in terms of this core. Hence, this formulation is representative of RULER.

Let $\mathcal{R}$ be a set of rule names. A *term* is a variable, value, or a function over terms (e.g. $x + 1$). A *rule expression* is a rule name r applied to a list of terms and is *pure* if these terms are function-free. A *premise* is an event, pure rule expression or guard, or a negation of any of these (we use ! for negation). A *rule term* is of the form $lhs \rightarrow rhs$ where lhs is a list of premises and rhs is a list of rule expressions. A *rule definition* is of the form $r(\overline{x})\{body\}$ where r is a rule name, $\overline{x}$ is a list of variables and $body$ is a set of rule terms. We call $r(\overline{x})\{body\}$ a rule definition for $r(\overline{x})$. Finally, A fact is a finite set of rule instances. A *rule instance* is a pair $\langle r, \theta \rangle$ where r is a rule name and θ is a valuation. We now define a rule system.

Definition 5 (Rule System). *A* rule system *is a tuple $\langle \mathcal{D}, \mathcal{B}, \mathcal{I} \rangle$ where $\mathcal{D}$ is a finite set of rule definitions, $\mathcal{B}$ is a finite set of* bad rule expressions *and $\mathcal{I}$ is an* initial *fact.*

A rule term $\mathit{lhs} \rightarrow \mathit{rhs}$ is *well-formed* if when the first occurrence of a variable in lhs is under a negation then this is its only occurrence in the rule term. A rule definition $\mathsf{r}(\overline{x})\{\mathit{body}\}$ is *well-formed* if every rhs in body only contains variables in $\overline{x}$ or the corresponding lhs. A rule system is *well-formed* if (i) all rule terms are well formed, (ii) there is at most one rule definition for each $r(\overline{x})$, and (iii) every rule expression used in rule terms has a corresponding definition. A rule instance $\langle r, \theta \rangle$ is well-formed for a rule system if there is a rule definition for $r(\overline{x})$ such that $\mathsf{dom}(\theta) = \overline{x}$. Below we assume a well-formed rule system of interest and will refer to its components directly.

The semantics of rule systems can be given in terms of a rewrite relationship on *facts*. Given a fact and an event we (i) find the set of rule instances in the fact that *fire*, and then (ii) update the fact with respect to these rule instances.

An *extended fact* is a finite set of rule instances and (ground) events. We define a *firing function* for extended fact Γ, valuation θ and premise as follows:

$$\begin{array}{lcl}
\mathsf{fire}(\Gamma, \theta, \mathbf{b}) & = & \theta \dagger \mathsf{match}(\mathbf{a}, \theta(\mathbf{b})) \quad \text{if} \quad \mathbf{a} \in \Gamma \wedge \mathsf{matches}(\mathbf{a}, \theta(\mathbf{b})) \\
\mathsf{fire}(\Gamma, \theta, r(\overline{x})) & = & \theta \dagger \mathsf{match}(\overline{v}, \theta(\overline{x})) \quad \text{if} \quad r(\overline{v}) \in \Gamma \wedge \mathsf{matches}(\overline{v}, \theta(\overline{x})) \\
\mathsf{fire}(\Gamma, \theta, \gamma) & = & \theta \quad \text{if} \quad \gamma(\theta) \\
\mathsf{fire}(\Gamma, \theta, !t) & = & \theta \quad \text{if} \quad \mathsf{fire}(\Gamma, \theta, t) = \bot \\
\mathsf{fire}(\Gamma, \theta, t) & = & \bot \quad \text{otherwise}
\end{array}$$

This computes the extension of θ that satisfies the premise using the given extended fact. The first two lines match against events and rule expressions, the third line checks guards, the fourth line deals with negation, and the last line handles the case where the constraints of previous lines do not hold. This is lifted to lists of premises as follows:

$$\mathsf{fire}(\Gamma, \theta, \epsilon) = \theta \qquad \mathsf{fire}(\Gamma, \theta, \mathit{prems}) = \mathsf{fire}(\Gamma, \mathsf{fire}(\Gamma, \theta, \mathsf{head}(\mathit{prems})), \mathsf{tail}(\mathit{prems}))$$

We say that a rule instance $\langle r, \theta \rangle$ *fires* in an extended fact Γ if $\mathsf{fire}(\Gamma, \theta, \mathit{lhs}) \neq \bot$ where $\mathit{lhs} \rightarrow \mathit{rhs}$ is in the body of the rule definition for $r(\mathsf{dom}(\theta))$.

Given a rule system and extended fact Γ, we define the set of ground rule expressions that result from a rule instance $\langle r, \theta \rangle$ firing as follows:

$$\mathsf{fired}(\langle r, \theta \rangle, \Gamma) = \{\theta'(\mathit{rhs}) \mid \mathit{lhs} \rightarrow \mathit{rhs} \in r(\mathsf{dom}(\theta)) \wedge \theta' = \mathsf{fire}(\Gamma, \theta, \mathit{lhs})\}$$

where we write $\mathit{lhs} \rightarrow \mathit{rhs} \in r(\mathsf{dom}(\theta))$ to mean that $\mathit{lhs} \rightarrow \mathit{rhs}$ is in the body of the rule definition of $r(\mathsf{dom}(\theta))$. As $\theta'(\mathit{rhs})$ is now ground we evaluate all functions to ensure that it is also pure e.g. $[x \mapsto 1](s(x+1)) = s(1+1) = s(2)$.

We define a rewrite relation $\Delta \xrightarrow{\mathbf{a}} \Delta'$ for facts Δ and Δ' and ground event $\mathbf{a}$. Let $\Delta' = (\Delta_{NF} \backslash \Delta_R) \cup \Delta_F$ where Δ_{NF} is the set of rule instances in Δ that do not fire in $\Delta \cup \{\mathbf{a}\}$ and Δ_F and Δ_R are the smallest facts such that:

$$\begin{array}{lll}
\langle r', [\overline{x} \mapsto \overline{v}] \rangle \in \Delta_F & \text{if} & \langle r, \theta \rangle \text{ fires in } \Delta \cup \{\mathbf{a}\} \text{ and } r'(\overline{v}) \in \mathsf{fired}(\langle r, \theta \rangle, \Delta \cup \{\mathbf{a}\}) \\
\langle r', [\overline{x} \mapsto \overline{v}] \rangle \in \Delta_R & \text{if} & \langle r, \theta \rangle \text{ fires in } \Delta \cup \{\mathbf{a}\} \text{ and } !r'(\overline{v}) \in \mathsf{fired}(\langle r, \theta \rangle, \Delta \cup \{\mathbf{a}\})
\end{array}$$

where $r(\overline{x})$ is defined in $\mathcal{D}$. This defines Δ_F as the new rule instances after rules are fired and Δ_R as the rule instances that need to be removed after rules are fired.

This rewrite relation is transitively extended to traces to produce a final fact $\mathcal{I} \xrightarrow{\tau} \Delta$, where $\mathcal{I}$ is the initial fact. This final fact is accepting if it does not contain a rule instance $\langle r, \theta \rangle$ such that $r(\mathsf{dom}(\theta)) \in \mathcal{B}$, the set of bad rule expressions.

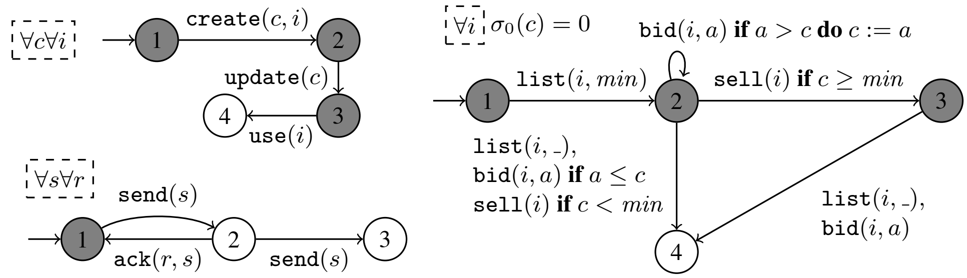


Fig. 1. QEA for (i) the *UnsafeIterator* property (top left), (ii) the *AuctionBidding* property (right), and (iii) the *Broadcast* property (bottom left).

3.3 Examples

We now introduce three example properties and specify them in the two languages. We will later use these to motivate, demonstrate, and discuss the translation. The three properties are:

- The *UnsafeIterator* property that an iterator i created from a collection c cannot be used after c is updated.
- The *AuctionBidding* property that after an item i is listed on an auction site with a reserve price min it cannot be relisted, all bids must be strictly increasing, and it can only be sold once this min price has been reached.
- The *Broadcast* property that for every sender s and receiver r, after s sends a message it should wait for an acknowledgement from r before sending again. Receivers are identified exactly as objects that acknowledge messages.

These are formalised as QEA in Figure 1 and as rule systems in Figure 2. One case that may require some explanation is the rule system for the *Broadcast* property. This needs to build up knowledge about the set of sender and receiver objects explicitly (whilst in trace slicing this is done implicitly), relying on the knowledge that the set of receivers must be fixed once a sender sends for the second time.

4 Translating Quantified Event Automata to Rule Systems

We now show how to produce a rule system from a QEA. This will consist of three translations on the QEA until it is in a form where we can apply a local translation of each state to a rule definition. The translation has been implemented in SCALA (see `https://github.com/selig/qea_to_rules`).

4.1 An equivalent representation with labelled states

We introduce an annotation of QEA that replaces states with *labelled states*. The idea is that a state will be labelled with the set of variables that are seen on all paths to that state. Let $\langle q, S\rangle$ be a *labelled state* where q is a state and S a (possibly empty) set of variables. Given a set of states Q and a set of variables X let $LS = Q \times 2^X$ be the (finite) set of labelled states.

$$\left\langle \begin{array}{rl} \mathsf{Start}\{ & \texttt{create}(c,i), !\mathsf{Unsafe}(c,i) \to \mathsf{Created}(c,i), \mathsf{Start}\ \} \\ \mathsf{Created}(c,i)\{ & \texttt{update}(c) \to \mathsf{Unsafe}(c,i)\ \} \\ \mathsf{Unsafe}(c,i)\{ & \texttt{use}(i) \to \mathsf{Fail}\ \} \end{array} , \{\mathsf{Fail}\}, \langle \mathsf{Start}, []\rangle \right\rangle$$

$$\left\langle \begin{array}{rl} \mathsf{Start} & \{\ \texttt{list}(i, min), !\mathsf{Live}(i,m), !\mathsf{Sold}(i) \to \mathsf{Live}(i, min, 0), \mathsf{Start}\ \} \\ \mathsf{Live}(i,m,c) & \left\{ \begin{array}{l} \texttt{bid}(i,a), a > c \to \mathsf{Live}(i,a) \\ \texttt{sell}(i), c \geq m \to \mathsf{Sold}(i) \\ \texttt{list}(i,\_) \to \mathsf{Fail} \\ \texttt{bid}(i,a), a \leq c \to \mathsf{Fail} \\ \texttt{sell}(i), c < m \to \mathsf{Fail} \end{array} \right\} \\ \mathsf{Sold}(i) & \left\{ \begin{array}{l} \texttt{list}(i,\_) \to \mathsf{Fail} \\ \texttt{bid}(i,a) \to \mathsf{Fail} \end{array} \right\} \end{array} , \{\mathsf{Fail}\}, \langle \mathsf{Start}, []\rangle \right\rangle$$

$$\left\langle \begin{array}{rl} \mathsf{Start} & \left\{ \begin{array}{l} \texttt{send}(s), !\mathsf{S}(s) \to \mathsf{S}(s), \mathsf{Start} \\ \texttt{send}(s), !\mathsf{S}(s), R(r) \to \mathsf{Unsafe}(r,s), \mathsf{Start} \\ \texttt{send}(s), \mathsf{S}(s) \to \mathsf{Fixed} \\ \texttt{ack}(r,s), !\mathsf{R}(r) \to \mathsf{R}(r), \mathsf{Start} \end{array} \right\} \\ \mathsf{Fixed} & \left\{ \begin{array}{l} \texttt{ack}(r,s), !\mathsf{R}(r) \to \mathsf{Fail} \\ \texttt{send}(s), !\mathsf{S}(s), \mathsf{R}(r) \to \mathsf{S}(s), \mathsf{Unsafe}(r,s), \mathsf{Fixed} \end{array} \right\} \\ \mathsf{S}(s) & \left\{ \begin{array}{l} \texttt{send}(s), \mathsf{R}(r) \to \mathsf{Unsafe}(r,s), \mathsf{S}(s) \\ \texttt{ack}(r,s'), !\mathsf{R}(r), s \neq s' \to \mathsf{Unsafe}(r,s), \mathsf{S}(s) \end{array} \right\} \\ \mathsf{Unsafe}(r,s) & \left\{ \begin{array}{l} \texttt{send}(s) \to \mathsf{Fail} \\ \texttt{ack}(r,s) \to \mathit{empty} \end{array} \right\} \\ \mathsf{R}(r) & \{\} \end{array} , \left\{ \begin{array}{l} \mathsf{Unsafe}(r,s), \\ \mathsf{Fail} \end{array} \right\}, \langle \mathsf{Start}, []\rangle \right\rangle$$

Fig. 2. Rule systems for (i) the *UnsafeIterator* property (top), (ii) the *AuctionBidding* property (middle), and (iii) the *Broadcast* property (bottom). Assuming general rule definition $\mathsf{Fail}\{\}$.

A QEA over labelled states is *well-labelled* if when $\langle q_2, S_2\rangle$ is reachable from $\langle q_1, S_1\rangle$ we have $S_1 \subseteq S_2$. The previous *Broadcast* QEA is not well-labelled as the initial state would have an empty set of labels but there is an incoming transition using r and s. The equivalent well-labelled version of this (corresponding to the result of the construction introduced next) is given in Figure 3 (top). We show how to construct a well-labelled QEA defined over labelled states from a standard QEA. Given QEA $\langle X, Q, \mathcal{A}(X \cup Y), \delta, \mathcal{F}, q_0, \sigma_0\rangle$ we construct $\langle X, LS, \mathcal{A}(X \cup Y), \delta', \mathcal{F}', \langle q_0, \{\}\rangle, \sigma_0\rangle$ where δ' and $\mathcal{F}'$ are defined as the smallest sets satisfying the following:

$$\begin{array}{ll} (\langle q, S\rangle, e(\overline{x}), \gamma, \alpha, \langle q', S \cup (\overline{x}\backslash Y)\rangle) \in \delta' & \text{if} \ \ (q, e(\overline{x}), \gamma, \alpha, q') \in \delta \\ (\langle q, S\rangle, e(\overline{x}), \neg(\gamma_1 \vee \ldots \vee \gamma_n), \mathsf{id}, \langle q, S \cup \overline{x}\backslash Y\rangle) \in \delta' & \text{for} \ \ e(\overline{x}) \in \mathcal{A}(X \cup Y) \\ & \ \text{and all} \ \ (q, e(\overline{x}), \gamma_i, \alpha, q') \in \delta \\ \langle q, S\rangle \in \mathcal{F}' & \text{if} \ \ q \in \mathcal{F} \ \text{ and } \ S = X \end{array}$$

where $S \subseteq X$. The second item requires explanation; this captures the case where no transition can be taken and thus an implicit self-loop is performed as these transitions may be between states with different captured variables. Note that if no transitions for

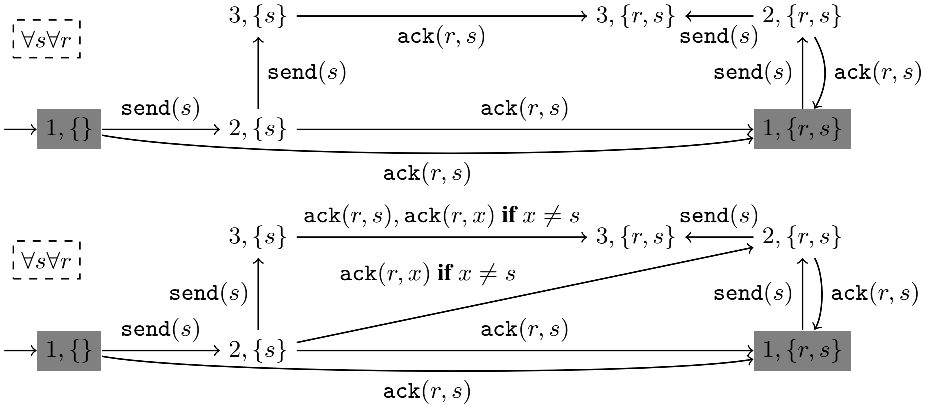


Fig. 3. *Well-labelled* and *domain-explicit* versions of the *Broadcast* QEA.

$e(\overline{x})$ exist then $\neg(\gamma_1 \vee \ldots \gamma_n)$ will be $true$. This may lead to unreachable states which can be safely removed. A special case of this would be where a guard becomes *false* by negating a $true$ guard. Note that final states must have the set of quantified variables X as their label. This fits with the observation that slicing only considers total valuations.

This resultant automaton over labelled states is equivalent to the original one as no new paths to final states are introduced and none are removed. From now on we will refer to QEA over labelled states as QEA if the labelling is clear from the context or unimportant. Additionally, we will assume all QEA are well-labelled.

4.2 A domain-explicit form

We make the following observation about the *Broadcast* property. Consider the trace $\texttt{send}(1).\texttt{ack}(2,3)$. After the first event the only (partial) valuation we can be aware of is $[s \mapsto 1]$. The second event extends the domain of r and requires us to consider $[s \mapsto 1, r \mapsto 2]$. However, $\texttt{ack}(2,3)$ is not relevant to $[s \mapsto 1]$. This will be problematic for our translation as in the rule system the decision about whether to extend a valuation must be made locally i.e. by making a transition. Here this can be resolved by adding a transition $(\langle 2, \{s\}\rangle, \texttt{ack}(r, x), x \neq s, \langle 2, \{r, s\}\rangle)$, which is one of two transitions added by the following construction as illustrated in Figure 3 (bottom). However, in general, we may need to add many similar transitions to capture all possible valuation extensions. We will now introduce an intermediate form that achieves this.

We introduce a conversion to *domain-explicit* QEA that will (i) ensure that ground events that extend an evaluation will always correspond to a transition in the automaton, but (ii) will also preserve the language of the QEA. To convert to domain-explicit form, for each labelled state $\langle q, S\rangle$ and event $e(\overline{x}) \in \mathcal{A}(X \cup Y)$ where $\overline{x} \cap (X/S) \neq \emptyset$ (it contains at least one new quantified variable) we add a set of transitions

$$(\langle q, S\rangle, \mathsf{e}(\overline{x}[x_i \mapsto \mathsf{fresh}(x_i)]), \bigwedge_{x \in R} x \neq \mathsf{fresh}(x), \mathsf{id}, \langle q, S \cup R\rangle)$$

where R is a non-empty subset of $S \cap (\overline{x}/Y)$ and $\mathsf{fresh}(x)$ produces a consistent fresh variable if $x \in R$ and x otherwise. These events are exactly those that will bind new quantified variables without needing to match the values of existing quantified variables. If $\overline{x}$ and S are disjoint then $\mathtt{e}(\overline{z}) = \mathtt{e}(\overline{x})$. Otherwise, a new event is created replacing one or more known quantified variables (in S) by a fresh unquantified variable along with a guard saying that the two are not equal.

The QEA resulting from this translation is well-labelled and equivalent (in terms of language accepted) to the original QEA. Equivalence is due to the fact that transitions are only created between copies of the same state, therefore no paths to final states are added or removed. Additionally, due to the skipping completion of QEA, adding events to the alphabet has no other side-effects.

4.3 A fresh-variable form

Our final translation on the QEA is to ensure that we can transform transitions in a QEA directly into a rule definition. Consider the transition $\langle\langle 2, \{i\}\rangle, \mathtt{bid}(i, a), \textbf{if } a > c, c := a, \langle 2, \{i\}\rangle\rangle$ from the labelled QEA for the *AuctionBidding* property (see preprint). We might try and write the following rule definition for this transition where we must include the set of unquantified variables Y in the parameters of the rule definition:

$$\mathsf{r}_2(i, min, c, a)\{\mathtt{bid}(i, a), a > c \rightarrow \mathsf{r}_2(i, min, a, a)\}$$

This is problematic as $\mathtt{bid}(i, a)$ will try and match this a with the a in the parameter list. To avoid this, we must replace instances of unquantified variables in transitions with fresh local versions. For example, this tranisition would become $\langle\langle 2, \{i\}\rangle, \mathtt{bid}(i, b), \textbf{if } b > c, a := b; c := a, \langle 2, \{i\}\rangle\rangle$ i.e. we replace a by b and then set $a := b$ in the assignment.

To perform this translation we replace each transition $\langle\langle q, S\rangle, e(\overline{x}), \gamma, \alpha, \langle q', S'\rangle\rangle \in \delta$ with the new transition for $y_i \in \overline{x} \cap Y$ and fresh z_i:

$$\langle\langle q, S\rangle, [y_i \mapsto z_i](e(\overline{x})), [y_i \mapsto z_i](\gamma), (z_i = y_i); \alpha, \langle q', S'\rangle\rangle$$

The resultant QEA is clearly equivalent as all paths remain the same.

4.4 The translation

Given a domain-explicit labelled QEA $\langle X, LS, \mathcal{A}(X \cup Y), \delta, \mathsf{F}, \langle q_0, \{\}\rangle, \sigma_0\rangle$ we construct a set of rule definitions $RD = \{r_q(S, Y) \mid \langle q, S\rangle \in LS\}$. The body for each rule definition is constructed by translating each transition starting at that state. The important step is knowing how to translate each transition based on whether the transition extends the label of quantified variables or not.

(i) Transitions with the same label. We first consider simple transitions that do not bind any new quantified variables. Let $(\langle q, S\rangle, e(\overline{x}), \gamma, \alpha, \langle q', S\rangle) \in \delta$ be such a transition. We introduce the following rule term for this transition

$$\mathtt{e}(\overline{x}), \gamma \rightarrow r_{q'}(S, \alpha(Y))$$

where we write $\alpha(Y)$ for the expansion of assignment α to Y e.g. $(x = y + 1)\{x, y\} = y + 1, y$. We shall call rule terms of this form kind (i).

(ii) Transitions extending the label. Recall that the small-step semantics for QEA depended on the principle of maximality. We need to reproduce this in the constructed rule system. The notion of maximality applies when a valuation is *extended* with information about new quantified variables and the extension is required only if there is no larger consistent valuation. For transition $(\langle q, S\rangle, e(\overline{x}), \gamma, \alpha, \langle q', S'\rangle) \in \delta$ where $S \subset S'$ we introduce the following rule term

$$\mathsf{e}(\overline{x}), \gamma, !r_1(S_1, Y_1), \ldots, !r_n(S_n, Y_n) \rightarrow r_{q'}(S', \alpha(Y)), r_q(S, Y)$$

for $r_i(S_i, Y) \in RD$, $S \subset S_i$, and fresh copies Y_i of Y. We treat assignment α as the valuation given by applying it to the identity valuation. We shall call rule terms of this form kind (ii). Two features of this rule term should be explained. Firstly, $!r_1(S_1), \ldots, !r_n(S_n)$ captures maximality as it states that there is no rule instance with a valuation larger than and consistent with the current one. Secondly, the two rule expressions on the right serve two separate purposes: $r_{q'}(S')$ is the new valuation in its new state and $r_q(S)$ is re-added as the initial valuation should stay in the current state.

As an example, the domain-explicit labelled QEA for the Broadcast property is translated to the following set of rule definitions (generated by our tool).

$$\begin{array}{rl}
\mathsf{r}_1 & \left\{ \begin{array}{l} \mathtt{ack}(r,s), !\mathsf{r}_1(r,s), !\mathsf{r}_2(r,s), !\mathsf{r}_2(s), !\mathsf{r}_3(r,s), !\mathsf{r}_3(s) \rightarrow \mathsf{r}_1, \mathsf{r}_1(r,s) \\ \mathtt{send}(s), !\mathsf{r}_1(r,s), !\mathsf{r}_2(r,s), !\mathsf{r}_2(s), !\mathsf{r}_3(r,s), !\mathsf{r}_3(s) \rightarrow \mathsf{r}_1, \mathsf{r}_2(s) \end{array} \right\} \\
\mathsf{r}_1(r,s) & \left\{ \mathtt{send}(s) \rightarrow \mathsf{r}_2(r,s) \right\} \\
\mathsf{r}_2(s) & \left\{ \begin{array}{l} \mathtt{send}(s) \rightarrow \mathsf{r}_3(s) \\ \mathtt{ack}(r,s_p), s \neq s_p, !\mathsf{r}_1(r,s), !\mathsf{r}_2(r,s), !\mathsf{r}_3(r,s) \rightarrow \mathsf{r}_2(s), \mathsf{r}_2(r,s) \\ \mathtt{ack}(r,s), !\mathsf{r}_1(r,s), !\mathsf{r}_2(r,s), !\mathsf{r}_3(r,s) \rightarrow \mathsf{r}_2(s), \mathsf{r}_1(r,s) \end{array} \right\} \\
\mathsf{r}_2(r,s) & \left\{ \begin{array}{l} \mathtt{send}(s) \rightarrow \mathsf{r}_3(r,s) \\ \mathtt{ack}(r,s) \rightarrow \mathsf{r}_1(r,s) \end{array} \right\} \\
\mathsf{r}_3(s) & \left\{ \begin{array}{l} \mathtt{ack}(r,s_p), s \neq s_p, !\mathsf{r}_1(r,s), !\mathsf{r}_2(r,s), !\mathsf{r}_3(r,s) \rightarrow \mathsf{r}_3(s), \mathsf{r}_3(r,s) \\ \mathtt{ack}(r,s), !\mathsf{r}_1(r,s), !\mathsf{r}_2(r,s), !\mathsf{r}_3(r,s) \rightarrow \mathsf{r}_3(s), \mathsf{r}_3(r,s) \end{array} \right\} \\
\mathsf{r}_3(r,s) & \{\}
\end{array}$$

We have now described how to produce a rule body for each rule definition by translating the transitions as described above. A rule system is the set $\mathcal{D}$ of rule definitions for each state in LS, the bad rule expressions $\mathcal{B} = \{r(S) \mid \langle q, S\rangle \notin \mathcal{F}\}$ and the initial state = $\{\langle r_{q_0}, \sigma_0\rangle\}$.

We can now state our theorem that the translation is correct i.e. it preserves the property defined by the QEA.

Theorem 1. *Given a domain-explicit $\mathcal{Q}$, let RS be the rule system given by the above translation. For monitoring state M_τ and rule state Δ_τ if*

$$M_\tau = \tau * [[] \mapsto \{\langle q_0, \sigma_0(Y)\rangle\}] \qquad \textit{and} \qquad \{\langle r_{q_0}, \sigma_0\rangle\} \xrightarrow{\tau} \Delta_\tau$$

then for any valuation θ

$$M_\tau(\theta) = \{\langle q, \sigma\rangle \mid \langle r_q, \theta \cup \sigma \cup \sigma'\rangle \in \Delta_\tau \wedge \mathsf{dom}(\sigma') \cap Y = \emptyset\}$$

The proof can be found in the appendix. The translation is *decidable*; any QEA of the form given in Section 3.1 can be translated to a rule system (which is neither unique nor minimal; no good notion of minimality exists). The size of the resulting rule system is potentially $O(|Q| \times 2^{|X|})$ due to the well-labelled translation introducing new states.

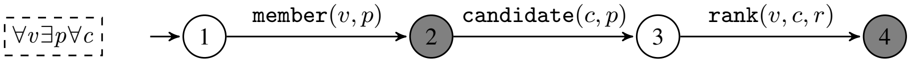


Fig. 4. A QEA for the *CandidateSelection* property taken from [3].

5 Discussion and Related Work

In this section we explore what we have learned about the relationship between the two languages introduced in Section 3 by the development of the previous translation. We consider the *expressiveness* of the languages, the *efficiency* of monitoring, how data is treated differently in each language, and the generality of our results.

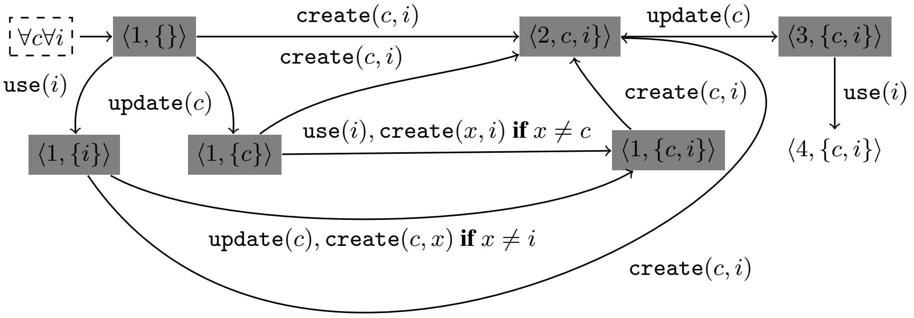
Expressiveness. Our translation shows that rule systems are at least as expressive as the form of QEA presented here (i.e. without existential quantification, see below). The remaining questions are whether they are strictly more expressive and what effect the choice of presentation for QEA has had on this translation. The first question can be answered positively. Our previous work [17] has given an example of a property that cannot be captured via trace slicing. This was a lock-ordering inspired property but the general form relied on second-order quantification to define a notion of reachability. For the second question we consider the differences in the presentation of QEA with [3].

- *Existential Quantification.* Existential quantification can be useful in certain cases but we do not yet know how to extend the translation to include it generally. For example, it is very difficult to write a rule system for the QEA given in Figure 4. It seems that it will be necessary to extend rule systems with additional support either via explicit quantification or a specialised notion of non-determinism that splits the state into multiple states where only one needs to be accepting. This property is formalised as a rule system in [17] but this relies on explicitly recording all facts and performing a computation on a special end of trace event.
- *Non-Determinism.* In [3], QEA were given some-path non-determinism but in [17] we observed that the most common use of non-determinism was to capture *negative* properties (the bad behaviour) and in this case all-path non-determinism is preferable. Hence, MARQ [24] supports both. To also support some-path non-determinism here (which is not commonly used) we would need to add branching and a notion of *good* facts to our rule systems (as is done in RULER).

Both existential quantification and non-determinism are rarely used features of QEA.

Efficiency. In this translation we are able to go from QEA, which have a highly efficient monitoring algorithm [6], to rule systems, which do not [16]. This appears to be the wrong direction to make gains in efficiency. However, we can make two observations that may lead to improvements in efficiency in both systems.

Firstly, let us consider the translation of the *UnsafeIter* property given in Figure 5 where we also give the explicit-domain labelled QEA. On inspection we can see that the rule definitions $\mathsf{r}_1(i)$, $\mathsf{r}_1(c)$, and $\mathsf{r}_1(c, i)$ are redundant as every trace that leads to a rule instance $\langle \mathsf{r}_2, \theta \rangle$ via these rules will also be produced if they are absent. This should



$$
\begin{array}{rl}
\mathsf{r}_1 & \left\{\begin{array}{l}
\mathtt{update}(c), !\mathsf{r}_1(c), !\mathsf{r}_1(i), !\mathsf{r}_1(c,i), !\mathsf{r}_2(c,i), !\mathsf{r}_3(c,i), !\mathsf{r}_4(c,i) \rightarrow \mathsf{r}_1, \mathsf{r}_1(c) \\
\mathtt{create}(c,i), !\mathsf{r}_1(c), !\mathsf{r}_1(i), !\mathsf{r}_1(c,i), !\mathsf{r}_2(c,i), !\mathsf{r}_3(c,i), !\mathsf{r}_4(c,i) \rightarrow \mathsf{r}_1, \mathsf{r}_2(c,i) \\
\mathtt{use}(i), !\mathsf{r}_1(c), !\mathsf{r}_1(i), !\mathsf{r}_1(c,i), !\mathsf{r}_2(c,i), !\mathsf{r}_3(c,i), !\mathsf{r}_4(c,i) \rightarrow \mathsf{r}_1, \mathsf{r}_1(i)
\end{array}\right\} \\
\mathsf{r}_1(c) & \left\{\begin{array}{l}
\mathtt{create}(c,i), !\mathsf{r}_1(c,i), !\mathsf{r}_2(c,i), !\mathsf{r}_3(c,i), !\mathsf{r}_4(c,i) \rightarrow \mathsf{r}_1(c), \mathsf{r}_2(c,i) \\
\mathtt{use}(i), !\mathsf{r}_1(c,i), !\mathsf{r}_2(c,i), !\mathsf{r}_3(c,i), !\mathsf{r}_4(c,i) \rightarrow \mathsf{r}_1(c), \mathsf{r}_1(c,i) \\
\mathtt{create}(c_p,i), c \neq c_p, !\mathsf{r}_1(c,i), !\mathsf{r}_2(c,i), !\mathsf{r}_3(c,i), !\mathsf{r}_4(c,i) \rightarrow \mathsf{r}_1(c), \mathsf{r}_1(c,i)
\end{array}\right\} \\
\mathsf{r}_1(i) & \left\{\begin{array}{l}
\mathtt{create}(c,i_p), i \neq i_p, !\mathsf{r}_1(c,i), !\mathsf{r}_2(c,i), !\mathsf{r}_3(c,i), !\mathsf{r}_4(c,i) \rightarrow \mathsf{r}_1(i), \mathsf{r}_1(c,i) \\
\mathtt{update}(c), !\mathsf{r}_1(c,i), !\mathsf{r}_2(c,i), !\mathsf{r}_3(c,i), !\mathsf{r}_4(c,i) \rightarrow \mathsf{r}_1(i), \mathsf{r}_1(c,i) \\
\mathtt{create}(c,i), !\mathsf{r}_1(c,i), !\mathsf{r}_2(c,i), !\mathsf{r}_3(c,i), !\mathsf{r}_4(c,i) \rightarrow \mathsf{r}_1(i), \mathsf{r}_2(c,i)
\end{array}\right\} \\
\mathsf{r}_1(c,i) & \left\{ \mathtt{create}(c,i) \rightarrow \mathsf{r}_2(c,i) \right\} \\
\mathsf{r}_2(c,i) & \left\{ \mathtt{update}(c) \rightarrow \mathsf{r}_3(c,i) \right\} \\
\mathsf{r}_3(c,i) & \left\{ \mathtt{use}(i) \rightarrow \mathsf{r}_4(c,i) \right\} \\
\mathsf{r}_4(c,i) & \{\}
\end{array}
$$

Fig. 5. Fully transformed QEA and corresponding rule system for the *UnsafeIterator* property.

not be surprising as if we remove these rule definitions the rule system becomes very similar to the one given in Section 3.3, only with the addition of maximality guards. By making some operations carried out by the slicing structure explicit, we can identify an inherent redundancy in this computation, which should lead to an optimisation of the monitoring algorithm for QEA. Formalising this redundancy both for rule systems and QEA remains further work.

Secondly, one hope for this translation was to identify a fragment of rule systems that are amenable to the efficient indexing-based monitoring algorithms used for QEA. After removing the redundancy identified above the first rule definition becomes

$$
\mathsf{r}_1 \left\{ \mathtt{create}(c,i), !\mathsf{r}_2(c,i), !\mathsf{r}_3(c,i), !\mathsf{r}_4(c,i) \rightarrow \mathsf{r}_2(c,i), \mathsf{r}_1 \right\}
$$

which, when compared to the rule system in Figure 2, includes additional negated rule expressions in the premises (which add to monitoring complexity). So taken ‘as is’ the resulting rule system is likely to be less efficient. However, these negated rule expressions give an explicit order in which to check rule definitions when matching incoming events (in a similar way to how indexing works for QEA) and it is plausible that this can be used to improve RULER’s monitoring algorithm by either detecting rule systems of this form or automatically checking if the given rule system is equivalent to a rule system of this form (as it is in this case). Therefore, the translation suggests a future direction for developing efficient indexing for rule-based runtime verification tools.

Treatment of Data. There are two main differences in the treatment of data that this work has highlighted. Firstly, QEA makes quantification domains implicit whereas rule systems make them explicit e.g. in QEA new bindings are produced by the monitoring algorithm whereas a rule needs to fire for a new binding in a rule system. This can have implications for readability – in rule systems it is somewhat easier to see what the domains are but in some circumstances having to encode these domains can make the actual behaviour difficult to understand. For example, the resulting rule system for the *Broadcast* property is much bigger than the original QEA. An advantage of making the domain explicit in rule systems is that domain knowledge can be used to ignore some part of the domain (as seen in the *UnsafeIterator* example above). This translation provides a mechanism for understanding exactly what the domain of quantification defined by a QEA is. Secondly, the use of maximality in trace-slicing hides a lot of operational details in the semantics – making this explicit in rule systems demonstrates the implicit work required to ensure that maximality is preserved. In some cases maximality is not necessary and this work can be removed in a rule system.

Generality. We now consider how general this translation is i.e. does it apply to all trace-slicing and rule-based approaches. The first system to use the trace-slicing idea was tracematches [1]. The use of suffix-based matching meant that the authors avoided the main technical difficulty in slicing i.e. dealing with partial valuations, which required maximality. Our translation does not work with suffix-matching but this could be encoded as another transformation on the QEA. The JAVAMOP system [22] has made the slicing approach popular with its efficient implementation. The QEA formalism [3, 23] was inspired by JAVAMOP. The notion of slicing presented here is compatible with that used in JAVAMOP as this also relies on *maximality*. Rule systems for runtime monitoring were introduced by the RULER tool [4, 2] and are used in TraceContract [5] and LOGFIRE [16] where a similar approach is taken i.e. a global set of instances or facts are rewritten by an associated set of rules. The rule systems described here can be considered a core subset of RULER and could be embedded into these other systems.

6 Conclusion

We have described the formal construction of a translation from the parametric trace slicing based QEA formalism to a rule system in the style of RULER. The translation has been shown to be equivalent to the small-step semantics for QEA.This translation gives insights into how parametric trace slicing and rule systems handle data differently. We observed that, to ensure the same property is described, it is necessary to (i) enforce complex maximality constraints on rule definitions, making them heavily interdependent, and (ii) add additional events and intermediate states to record the possible valuations as they are created. We have implemented the translation as a SCALA program. This will allow us to explore further optimisations of the translation, for example, by identifying redundant intermediate states and performing a backwards-analysis to introduce unquantified variables when they are first needed (the *AuctionBidding* translation would benefit from this). We are also looking at formalising this work in a proof assistant to give more rigorous guarantees of its correctness. In our general work on exploring the relationships between specification languages for runtime verification our next step will be to translate rule systems into a first-order temporal logic.

References

1. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, October 2005.
2. 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.
3. Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, pages 68–84, 2012.
4. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.
5. Howard Barringer and Klaus Havelund. Tracecontract: a Scala DSL for trace analysis. In *Proc. of the 17th international conference on Formal methods*, pages 57–72, Berlin, Heidelberg, 2011.
6. Ezio Bartocci, Borzoo Bonakdarpour, Yliès Falcone, Christian Colombo, Normann Decker, Felix Klaedtke, Klaus Havelund, Yogi Joshi, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First international competition on runtime verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 2017.
7. Ezio Bartocci, Ylies Falcone, Adrian Francalanza, Martin Leucker, and Giles Reger. An introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–23. Springer, 2018.
8. David Basin, Mat Harvan, Felix Klaedtke, and Eugen Zlinescu. Monpoly: Monitoring usage-control policies. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 360–364. Springer Berlin Heidelberg, 2012.
9. Andreas Bauer, Jan-Christoph Kster, and Gil Vegliach. The ins and outs of first-order runtime verification. *Formal Methods in System Design*, pages 1–31, 2015.
10. Laura Bozzelli and César Sánchez. Foundations of boolean stream runtime verification. *Theoretical Computer Science*, 631:118–138, 2016.
11. Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.
12. Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime monitoring of synchronous systems. In *Proc. of the 12th Int. Symposium on Temporal Representation and Reasoning*, pages 166–174, 2005.
13. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014.*, pages 341–356, 2014.
14. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In Manfred Broy and Doron Peled, editors, *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear*. IOS Press, 2013.
15. Sylvain Hallé and Raphaël Khoury. Runtime monitoring of stream logic formulae. In *Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers*, pages 251–258, 2015.
16. Klaus Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):143–170, 2015.

17. Klaus Havelund and Giles Reger. Specification of parametric monitors. In *Formal Modeling and Verification of Cyber-Physical Systems, 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*, pages 151–189, 2015.
18. Klaus Havelund and Giles Reger. Runtime verification logics - a language design perspective. In *KIMfest 2017*. Springer, 2017.
19. Klaus Havelund, Giles Reger, Eugen Zalinescu, and Daniel Thoma. Monitoring events that carry data. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 60–97. Springer, 2018.
20. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2008.
21. Ramy Medhat, Yogi Joshi, Borzoo Bonakdarpour, and Sebastian Fischmeister. Parallelized runtime verification of first-order LTL specifications. Technical report, University of Waterloo, 2014.
22. Patrick Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *J Software Tools for Technology Transfer*, pages 1–41, 2011.
23. Giles Reger. *Automata Based Monitoring and Mining of Execution Traces*. PhD thesis, University of Manchester, 2014.
24. Giles Reger, Helena Cuenca Cruz, and David Rydeheard. MARQ: monitoring at runtime with QEA. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, 2015.
25. Giles Reger, Sylvain Hallé, and Yliès Falcone. Third international competition on runtime verification CRV 2016. In *RV 2016*, 2016.
26. Giles Reger and David Rydeheard. From first-order temporal logic to parametric trace slicing. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*, pages 216–232, Cham, 2015. Springer International Publishing.
27. 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.

A Correctness of the translation

Here we show that the translation of Section 4 is correct i.e. the QEA and rule system accept the same traces. We first consider the case where $Y = \emptyset$ i.e. there are no unquantified variables and therefore the valuations in rule instances correspond directly to valuations in the domain of the monitoring state from the QEA semantics.

Lemma 1. *Let Δ be a fact of a rule system produced by the translation of a* QEA *with $Y = \emptyset$ and let $\mathbf{a}$ be a ground event. The rule instance $\langle r_i, \theta_i \rangle \in \Delta$ will fire for $\mathbf{a}$ with a rule term of kind (ii) to produce $\langle r_k, \theta_k \rangle$ if and only if $\theta_k \in \mathsf{extensions}(\theta_i, \mathbf{a})$ for the domain-explicit labelled form of the* QEA.

Proof. Firstly, for the domain-explicit form $\mathsf{from}(\mathbf{a})$ in the QEA semantics can be replaced by the equivalent

$$\{\mathsf{match}(\mathbf{a}, \mathbf{b}) \mid \exists \mathbf{b} \in \mathcal{A}(X \cup Y) : \mathsf{matches}(\mathbf{a}, \mathbf{b})\}$$

as one no longer needs to consider submaps of the matching valuation as the domain-explicit form extends the alphabet to ensure there will be an event which, when matched with, will produce this submap.

The set $\mathsf{extensions}(\theta_i, \mathbf{a})$ is then given by extending θ by consistent valuations in $\mathsf{from}(\mathbf{a})$. The domain-explicit form ensures that whenever there is a state that is not labelled with all quantified variables then there will be a transition for every event $\mathbf{b}$ that could extend the label. Therefore, there will be a rule of kind (ii) for every such event and these will fire to produce exactly the valuations in $\mathsf{extensions}(\theta_i, \mathbf{a})$.

Lemma 2. *Let Δ be a fact of a rule system produced by the translation. If $\langle r_i, \theta_i \rangle \in \Delta$ fires with a rule term of kind (ii) to produce $\langle r_k, \theta_k \rangle$ then $\mathsf{maximal}(\Delta, \theta_k) = \theta_i$.*

Proof. We assume that $\langle r_i, \theta_i \rangle \in \Delta$ fires with a rule term of kind (ii) to produce $\langle r_k, \theta_k \rangle$ and show that $\mathsf{maximal}(\Delta, \theta_k) = \theta_i$. Recall that $\mathsf{maximal}(\Delta, \theta_k) = \theta_i$ iff

$$\theta_i \in \Delta \wedge \theta_i \sqsubseteq \theta_k \wedge \forall \theta' \in \Delta : \theta' \sqsubseteq \theta_k \Rightarrow \theta_i \not\sqsubset \theta'$$

The first conjunct holds by definition and the second conjunct follows from the rule term being of kind (ii) (i.e. $S \subset S'$). The last part can be shown by contradiction. Assume $\langle r', \theta' \rangle \in \Delta$ such that $\theta' \sqsubseteq \theta_k$ and $\theta_i \sqsubset \theta'$. This means that $\mathsf{dom}(\theta_i) \subset \mathsf{dom}(\theta')$ and therefore one of the terms $!r_1(S_1), \ldots, !r_n(S_n)$ in the corresponding kind (ii) rule term is false and would not fire for $\langle r_i, \theta_i \rangle$, a contradiction.

We use these two lemmas to establish equivalence of the domain-explicit form and its translation. We have already argued for the language preservation of the transformations to domain-explicit form.

Theorem 2. *Given a domain-explicit labelled* QEA*, let RS be the rule system given by the above translation. For monitoring state M_τ and fact Δ_τ if*

$$M_\tau = \tau * [[] \mapsto \{q_0\}] \qquad \textit{and} \qquad \{\langle r_{q_0}, [] \rangle\} \xrightarrow{\tau} \Delta_\tau$$

then for any valuation θ

$$M_\tau(\theta) = \{q \mid \langle r_q, \theta \rangle \in \Delta_\tau\}$$

Proof. By induction on τ. The base case is trivial as initially they both only contain the initial state. For $\tau = \tau'.\mathbf{a}$ we have

$$M_{\tau'}(\theta) = \{q \mid \langle r_q, \theta \rangle \in \Delta_{\tau'}\}$$

as our induction principle. As there must be the same valuations in $M_{\tau'}$ and $\Delta_{\tau'}$ we will show that for any such valuation θ the effects on M_τ and Δ_τ are the same. That is (i) the information related to θ in each structure is updated in the same way and (ii) the additional valuations added from θ due to $\mathbf{a}$ are the same.

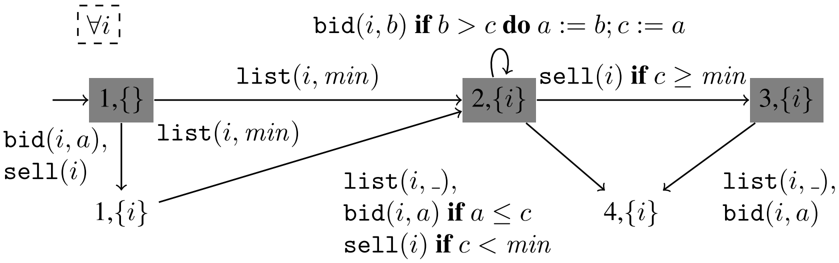
Let us take (i) first. According to Definition 4 if $\mathbf{a}$ is not relevant then no rules are fired, which matches with no transitions being taken in the single step construction. If $\mathbf{a}$ is relevant then $M_\tau(\theta) = \mathsf{next}(M_{\tau'}(\theta), \mathbf{a}, \theta)$. We can consider each $q \in M_{\tau'}(\theta)$ separately. For q we know $\langle r_q, \theta \rangle \in \Delta$. We argue that the firing rule terms of $r_q(\mathsf{dom}(\theta))$ match the result of $\mathsf{next}(M_{\tau'}(\theta), \mathbf{a}, \theta)$. A condition of next is that no new quantified variables are bound, this corresponds to kind (i) rule terms. As there is a one-to-one correspondence between transitions and rule terms then if a transition is taken the corresponding rule will fire and vice versa. As per the definition of next, if no transitions can be taken then the state remains the same, similarly if no kind (i) rule terms are fired then either a kind (ii) rule term is fired and the rule instance persists or no rule terms fire and the rule instance persists.

Now let us consider (ii). There are two parts, (a) exactly the same valuations are added and (b) the new valuations are in the same states. The first part (a) follows from Lemma 1 as the set of valuations added in both cases is the same. The second part (b) follows from Lemma 2 as in both cases it is the maximal valuation that is used to define which states are associated with the new valuation.

We will now sketch how this proof needs to be modified to take account of QEA with non-empty Y. This can be used to establish Theorem 1 stated earlier. The only difference in the above proofs is that the valuations in rule instances must now be split into parts for quantified variables X and unquantified variables Y. The second part plays no part in the proof as the proofs of Lemmas 1 and 2 as these are solely about relating the domain of the monitoring state to the quantified parts of rule instances. In the proof of Theorem 2 we need to convince ourselves that this part of the valuation is correctly treated (i.e. equivalently to the next function). This follows trivially from the fact that the same assignment is applied directly in both cases.

B Translation of Auction Bidding Property

The transformed QEA for this property is given as:



The resulting rule definitions are then

$$
\begin{aligned}
&\mathsf{r}_1(m, c, a) \left\{ \begin{array}{l}
\texttt{list}(i, n), !\mathsf{r}_1(i, m_1, c_1, a_1), !\mathsf{r}_2(i, m_2, c_2, a_2), !\mathsf{r}_3(i, m_3, c_3, a_3), \\
\qquad !\mathsf{r}_4(i, , m_4, c_4, a_4) \rightarrow \mathsf{r}_2(i, n, c, a), \mathsf{r}_1(m, c, a) \\
\texttt{bid}(i, b), !\mathsf{r}_1(i, m_1, c_1, a_1), !\mathsf{r}_2(i, m_2, c_2, a_2), !\mathsf{r}_3(i, m_3, c_3, a_3), \\
\qquad !\mathsf{r}_4(i, , m_4, c_4, a_4) \rightarrow \mathsf{r}_1(i, m, c, a), \mathsf{r}_1(m, c, a) \\
\texttt{sell}(i), !\mathsf{r}_1(i, m_1, c_1, a_1), !\mathsf{r}_2(i, m_2, c_2, a_2), !\mathsf{r}_3(i, m_3, c_3, a_3), \\
\qquad !\mathsf{r}_4(i, , m_4, c_4, a_4) \rightarrow \mathsf{r}_1(i, m, c, a), \mathsf{r}_1(m, c, a)
\end{array} \right\} \\
&\mathsf{r}_1(i, m, c, a) \; \{ \; \texttt{list}(i, n) \rightarrow \mathsf{r}_2(i, n, c, a), \mathsf{r}_1(m, c, a) \; \} \\
&\mathsf{r}_2(i, m, c, a) \left\{ \begin{array}{l}
\texttt{bid}(i, b), b > c \rightarrow \mathsf{r}_2(i, m, b, b) \\
\texttt{sell}(i), c \geq m \rightarrow \mathsf{r}_3(i, m, c, a) \\
\texttt{list}(i, x) \rightarrow \mathsf{r}_4(i, m, c, a) \\
\texttt{bid}(i, b), b < c \rightarrow \mathsf{r}_4(i, m, c, a) \\
\texttt{sell}(i), c < m \rightarrow \mathsf{r}_4(i, m, c, a)
\end{array} \right\} \\
&\mathsf{r}_3(i, m, c, a) \left\{ \begin{array}{l}
\texttt{list}(i, x) \rightarrow \mathsf{r}_4(i, m, c, a) \\
\texttt{bid}(i, b) \rightarrow \mathsf{r}_4(i, m, c, a)
\end{array} \right\} \\
&\mathsf{r}_4(i, m, c, a) \; \{ \; \}
\end{aligned}
$$