International Journal on Software Tools for Technology Transfer (2021) 23:209–228
https://doi.org/10.1007/s10009-021-00608-0

GENERAL

Special Issue: RV 2018

From parametric trace slicing to rule systems

Giles Reger[1] · David Rydeheard[1]

Accepted: 12 January 2021 / Published online: 27 February 2021

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. This paper extends a previous conference version of this paper with further examples, a proof of correctness, and an optimisation based on a notion of redundancy observed during the development of the translation.

Keywords Runtime verification · Runtime monitoring · Parametric trace slicing · Rule systems

1 Introduction

Runtime verification [9,17,18,27] 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. Recent work has attempted to place some structure on this work in the form of a taxonomy [18], which has clarified different approaches, but leaves open the question of how they are related.

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 different languages that have been introduced. We believe that the field can be considerably improved by a better understanding of this space. Indeed, a contribution of this paper is an optimisation of an existing monitoring technique inspired by observations made during our work to understand the relationship between existing techniques.

This paper extends a previous version [35] that appeared in the 18th International Conference on Runtime Verification. It specifically explores the relationship between two particular approaches to specification for parametric runtime verification: *parametric trace slicing* and *rule systems*. As discussed later, we have chosen these two languages as they represent significant points in the space of runtime verification languages—parametric trace slicing is the key technology behind the JAVAMOP [29] and QEA languages [4,30] and RULER [3], TRACECONTRACT [6], and LOGFIRE [20] all take a rule-based approach.

Here we describe the main contributions of the paper and identify extensions to the previous paper. We begin by describing the general setting we are working in (Sect. 2) before introducing these two languages (Sects. 3–6). The main contribution of the paper is a translation from specifications using parametric trace slicing to those using rules (Sect. 7). These sections (Sects. 2–7) have been expanded

✉ Giles Reger
giles.reger@manchester.ac.uk

[1] University of Manchester, Manchester, UK

with additional explanation and extended examples. We then introduce a new detailed proof of correctness (Sect. 8) and a new optimisation of the translation (Sect. 9). This optimisation is based on a new notion of redundancy that is inspired by observations made during the earlier translation work. Importantly, this newly identified general notion of redundancy subsumes existing notions in the literature. 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 (Sect. 10).

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 (traces) via some instrumentation method. This is a common starting point, and techniques for extracting traces are described elsewhere [9].

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 things 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 $e(v_1, \ldots, v_n)$. An event is *ground* if it does not contain variables, and it is *propositional* if it does not contain any parameters. 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, $\texttt{login}(x, 42)$ is an event where x is a variable and 42 a value; the finite sequence $\texttt{login}(a, 42).\texttt{logout}(a)$ is a trace; and the set $\{\texttt{login}(a, 42).\texttt{logout}(a), \texttt{login}(b, 42).\texttt{logout}(b)\}$ is a property. We sometimes 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 are propositional; 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 [24] for an overview and [18] for how these fit into the general taxonomy of approaches). Parametric trace slicing [4,14,29] separates the issue of quantification from trace checking using a notion of *projection*. First-order extensions to temporal logic [10,11,16,28,36] rely on the standard logical treatment of quantification, introducing (somewhat complex) monitor construction techniques to handle this. Rule systems [3,6,21] and stream processing [12,15,19] 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 in 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. These are important variations, which we will consider further in future work, but they have their basis in the above stated problem and we are confident that much of our work can translate to these variations.

Our research question

Given this large space of specification languages, our fundamental research question (which we have been considering since 2010) 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 [8,33] 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 [22–24]. 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 [34] 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 Preliminaries

We introduce some additional technical definitions. If τ is a trace let τ_i be the ith event and $\tau|_n$ be the prefix of τ of length n. Let an *event alphabet* $\mathcal{A}(Z)$ be a set of events using variables in Z, e.g. for $Z = \{x\}$, $\mathcal{A}(\{x\})$ 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:

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

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 in a standard way, e.g. $e(x, y)[x \mapsto z] = e(z, y)$ and $(x \neq y)[x \mapsto z] = (z \neq y)$.

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, for example, $\mathit{Assign}(\{x, y, z\})$, contain assignments such as $x := y + 1; y := 0$ and $x = \mathsf{max}(y, z); y := y + 1$. 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.

4 Running examples

In this paper we will use three running examples to demonstrate the two languages in the next two sections and then motivate and discuss the translation later. These properties have been previously used as examples in previous work exploring specification languages for runtime verification [22–24]. The properties are as follows:

UnsafeIterator This is the property that an iterator i created from a collection c cannot be used after c is updated. An example trace of interest is

$$\mathtt{create}(C, I1).\mathtt{use}(I1).\mathtt{create}(C, I2).\mathtt{use}(I1).\ \mathtt{update}(C).\mathtt{use}(I2)$$

where a collection C is used to create two iterators $I1$ and $I2$. This trace violates the property as iterator $I2$ is used after C is updated. This property is interesting as it is about the

relationship between two objects, requiring two quantified variables.

AuctionBidding This is the property that after an item i is listed on an auction site with a reserve price *min* it cannot be re-listed, all bids must be strictly increasing, and it can only be sold once this *min* price has been reached. An example trace of interest involving two items *hat* and *ball* is

$$\texttt{list}(\mathit{hat}, 10).\texttt{bid}(\mathit{hat}, 5).\texttt{list}(\mathit{ball}, 4).\texttt{bid}(\mathit{ball}, 4).$$
$$\texttt{bid}(\mathit{ball}, 4).\texttt{sell}(\mathit{hat})$$

This trace violates the property for two reasons—bids on *ball* are not strictly increasing and *hat* is sold before it reaches its reserve price. This property is interesting as it makes use of local variables and guards and assignments on them.

Broadcast This is the property taken from the context of entities, e.g. autonomous vehicles, participating in a basic synchronisation mechanism where broadcast messages need to be acknowledged by all parties before continuing. More formally, 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. A trace of interest is

$$\texttt{send}(A).\texttt{send}(B).\texttt{ack}(B, A).\texttt{ack}(C, A).\texttt{ack}(C, B).$$
$$\texttt{send}(A).\texttt{send}(B)$$

which violates the property as B sends before their previous message is acknowledged by A. This property is interesting as it has looping behaviour (as well as quantifying over two objects) which requires extra effort to handle in the translation.

5 Parametric trace slicing with quantified event automata

In this section and the next, we will introduce two languages for describing parametric properties. The first is based on a notion of *parametric trace slicing* [14], introduced as a technique that transforms a monitoring problem involving quantification *over finite (but unknown) 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.

To illustrate this idea, consider a simply property *every thread should start and later end*. The propositional formulation is straightforward, e.g. all traces should match the regular expression $\mathsf{start}\ .^*\ \mathsf{end}$. To lift this to the parametric case with parametric trace slicing, we add a thread variable, e.g. $\mathsf{start}(x)\ .^*\ \mathsf{end}(x)$, and for each valuation, e.g. $[x \mapsto 1]$, we ground the specification, e.g. $\mathsf{start}(1)\ .^*\ \mathsf{end}(1)$, and then take the parametric trace, e.g. $\mathsf{start}(1).\mathsf{start}(2).\mathsf{end}(2).\mathsf{end}(1)$, and project it with respect to $[x \mapsto 1]$, e.g. $\mathsf{start}(1).\mathsf{end}(1)$ and check the projected trace against the grounded (now propositional) specification, e.g. $\mathsf{start}(1).\mathsf{end}(1) \stackrel{?}{\in} \mathcal{P}(\mathsf{start}(1)\ .^*\ \mathsf{end}(1))$. A key part of the parametric trace slicing approach is that the finite domain being quantified over is taken as the relevant values in the given trace as these can be taken as the finite subset of values of interest.

Structure Quantified event automata (QEA) [4] is a slicing-based formalism that generalises previous work on parametric trace slicing. We consider a restricted form[1] of QEA that does not allow existential quantification (as discussed in Sect. 10).

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 (hence Y are the remaining unquantified, local 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, e.g. they are free variables that are updated during the processing of a trace. 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. Note that the initial valuation does not appear in the original work of Barringer et al. [4] but has been added later as its utility became clear.

Examples Figure 1 presents the three running example properties as QEAs in graphical form[2]. Their meaning should become clear as the semantics is introduced in the next section, but note that, in this presentation, if a transition cannot be taken the automata stays in the current state (as opposed to transitioning to an implicit failure state). For example, state 2 of the *UnsafeIterator* QEA allows both `create` and `use` events. Note that *true* guards and *identity* assignments are omitted. Final states are indicated as grey states, whereas non-final states are white.

[1] This restriction is reasonable as usage of existential quantification (that cannot be replaced by free variables) is rare.

[2] Elsewhere [22] similar properties are given in a textual ASCII format. It has been commented that presenting QEAs graphically is 'cheating' from a specification language perspective as this is not how they are written by a user. However, for ease of comprehension here we choose the graphical presentation.

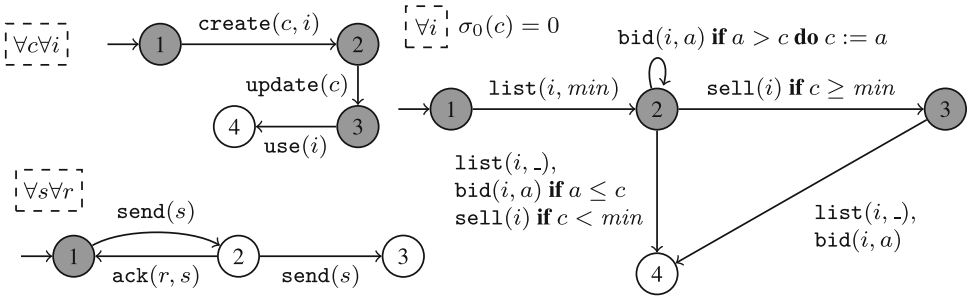


Fig. 1 QEA for (i) the *UnsafeIterator* property (top left), (ii) the *AuctionBidding* property (right), and (iii) the *Broadcast* property (bottom left)

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 this would be distracting here and the later material only relies on the small-step presentation. We refer the reader to other texts for this [4,30]. 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 set containing $(q', \alpha(\sigma \dagger \mathsf{match}(\mathbf{a}, \mathbf{b}, Y))$ if and only if

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

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.

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

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 extend an existing valuation θ given a new ground event $\mathbf{a}$ can be defined by:

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

This constructs all valuations that can be built directly and then uses the consistent (non-empty) 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 $i \in [1, m]$.

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

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

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

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 (i) it is the property that relates the small-step semantics to the big-step semantics (described elsewhere), and (ii) it makes the later translation complicated.

Quantification Domain It may not be obvious, but the above 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 QEALet $M_\tau = \tau * [\ \bot \mapsto \{(q_0, \sigma_0(Y))\}]$ 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. The **only** means that, in the case of non-determinism (which is rarely used), all paths that a trace can take through a QEA must end in a final state.

Illustrating Semantics We illustrate the above semantics using the *UnsafeIterator* property as this illustrates the notion of maximality. We will check whether the example trace τ from Sect. 4 is accepted by the QEA given in Fig. 1(i). As the QEA is deterministic and $Y = \emptyset$ we will simplify the presentation by representing the set of configurations by a single state instead. Hence, we start with the empty monitoring construct $[\bot \mapsto 1]$.

On the event $\texttt{create}(C, I1)$ we extend $\bot \mapsto 1$ with three new valuations. It might seem odd that we need $[i \mapsto I1]$ and $[c \mapsto C]$. However, these are necessary for the completeness of the approach. Later (Sect. 9) we will see that in this case they are redundant). Only the valuation $[c \mapsto C, i \mapsto I1]$ has an updated *next* state. The result is:

$$M_{\tau|_1} = \begin{bmatrix} \bot & \mapsto 1 \\ [c \mapsto C] & \mapsto 1 \\ [i \mapsto I1] & \mapsto 1 \\ [c \mapsto C, i \mapsto I1] & \mapsto 2 \end{bmatrix}$$

On $\texttt{use}(I1)$ the monitoring state remains unchanged as it already contains $[i \mapsto I1]$ and the set of next configurations for $[i \mapsto I1]$ and $[c \mapsto C, i \mapsto I1]$ are empty for this event. On the event $\texttt{create}(C, I2)$ we extend the monitoring state in a similar way to before. The $\texttt{update}(C)$ event then causes the next states for both of the total valuations to change, giving us the following monitoring state:

$$M_{\tau|_5} = \begin{bmatrix} \bot & \mapsto 1 \\ [c \mapsto C] & \mapsto 1 \\ [i \mapsto I1] & \mapsto 1 \\ [i \mapsto I2] & \mapsto 1 \\ [c \mapsto C, i \mapsto I1] & \mapsto 3 \\ [c \mapsto C, i \mapsto I2] & \mapsto 3 \end{bmatrix}$$

Finally, the event $\texttt{use}(I2)$ is relevant to $[i \mapsto I2]$ and $[c \mapsto C, i \mapsto I2]$ but only produces a new state for the latter; in this case the non-final state 4 is reached. Now, at the end of the trace, there is one total valuation mapped to a non-final state and, therefore, the trace is not in the property of the *UnsafeIterator* QEA.

Comparison with JavaMOP Given the close connection, it is worth quickly outlining the main differences between the approach taken in QEA and the parametric trace slicing utilised in JavaMOP. Firstly, JavaMOP supports a range of propositional languages that can be used in conjunction with slicing whilst QEA has an explicit automata-language. Secondly, there is no formal notion of 'local' variables in JavaMOP with such computation typically being achieved within code triggered by events. Finally, and importantly, there is no formal notion of quantification or property in the previous work on parametric trace slicing. Instead the focus is on triggering actions on when certain verdicts (or states in a state machine) are reached. These actions can be further triggered by restrictions on the valuations (of implicitly quantified variables) that they are associated with. In QEA there is a requirement for all such valuations to be *total* (e.g. bind all variables in X), but in JavaMOP partial valuations are allowed and they all support restricting attention to so-called *connected* valuations where each pair values co-occurs in an event in the trace. As discussed later, JavaMOP has also introduced a range of optimisations related to parametric trace slicing that allow it to monitor efficiently.

6 A rule-based approach

We now introduce an approach first introduced in RuleR [3] 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 [3], i.e. the extensions in [3] are either trivial or can be defined in terms of this core. Hence, this formulation is representative of RuleR whilst being simple

enough to present concisely. The presentation is similar to the RULER-lite variation described in [17].

The general idea behind the rule-based approach is that we have *facts* that are *rewritten* by a set of *rules* using an incoming event, with acceptance based on the facts that are (or are not) present in the final set of facts. In the nomenclature of the RULER system, a fact consists of a set of *rule instances*—each 'rule' is associated with a set of variables and each fact is an instance of one of the rules. Each 'rule' has its own set of terms of the form $lhs \rightarrow rhs$ with which its instances may be rewritten. When rules have empty variable sets and the *lhs* of rule terms consist of single events, this presentation coincides with that of state machines. However, rule systems can capture much more expressive languages by recording data in rule instances and having complex guards on rule terms, e.g. presence of absence of other rule instances.

Structure 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 $lhs \rightarrow rhs$ 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 $\mathsf{r}(\overline{x})\{body\}$ 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 $\mathsf{dom}(\theta) = \overline{x}$.

Examples Figure 2 presents the three running example properties as rule systems. The approach taken to specify the Start rule is a common idiom in rule systems to keep a rule 'alive' by reintroducing the rule when it fires. Indeed, RULER introduced special modifiers for such rules, which we omit for conciseness. As we will see later, this looping start rule can be used to emulate the quantifiers of QEA. 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. Finally, note the use of an undefined Fail rule which is often used to capture the 'other' cases. The RULER system had a notion of *asserted* rules that could capture implicit failure, but we omit this is as it can be encoded by the use of this Fail rule.

Semantics Here we assume a well-formed rule system of interest and will refer to its components directly, e.g. its rule definitions. The semantics of rule systems can be given in terms of a rewrite relationship on *facts* . Given a fact and an event we

1. Find the set of rule instances in the fact that *fire*, and then
2. Update the fact with respect to these rule instances.

For ease of presentation, we first introduce the notion of an *extended fact* as a finite set of rule instances and (ground) events. This allows us to add the incoming event to the current fact, to produce an extended fact, and define the operations directly on extended facts.

Firing Rule Instances To identify the firing rule instances we need to find those rule instances whose associated rule definition has a rule term with a premise list that can be satisfied in the current extended fact. This will likely involve extending the valuation associated with the rule instance. This extended valuation is required when producing new rule instances. Therefore, we define a *firing function* that computes this extended valuation first for a single premise and then lifted to premise lists.

For extended fact Γ, valuation θ and a single premise (e.g. event, rule expression, guard, or negated premise) we define fire as follows:

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

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:

$$
\begin{aligned}
\mathsf{fire}(\Gamma, \theta, \epsilon) &= \theta \\
\mathsf{fire}(\Gamma, \theta, \mathit{prems}) &= \mathsf{fire}(\Gamma, \mathsf{fire}(\Gamma, \theta, \mathsf{head}(\mathit{prems})), \mathsf{tail}(\mathit{prems}))
\end{aligned}
$$

Given a rule instance $\langle r, \theta \rangle$ let $\mathsf{find}(r, \theta)$ be the rule definition $r(\overline{x})\{body\}$ for $\overline{x} = \mathsf{dom}(\theta)$. We say that a rule instance $\langle r, \theta \rangle$ *fires* in an extended fact Γ if $\mathsf{fire}(\Gamma, \theta, lhs) \neq \bot$ where

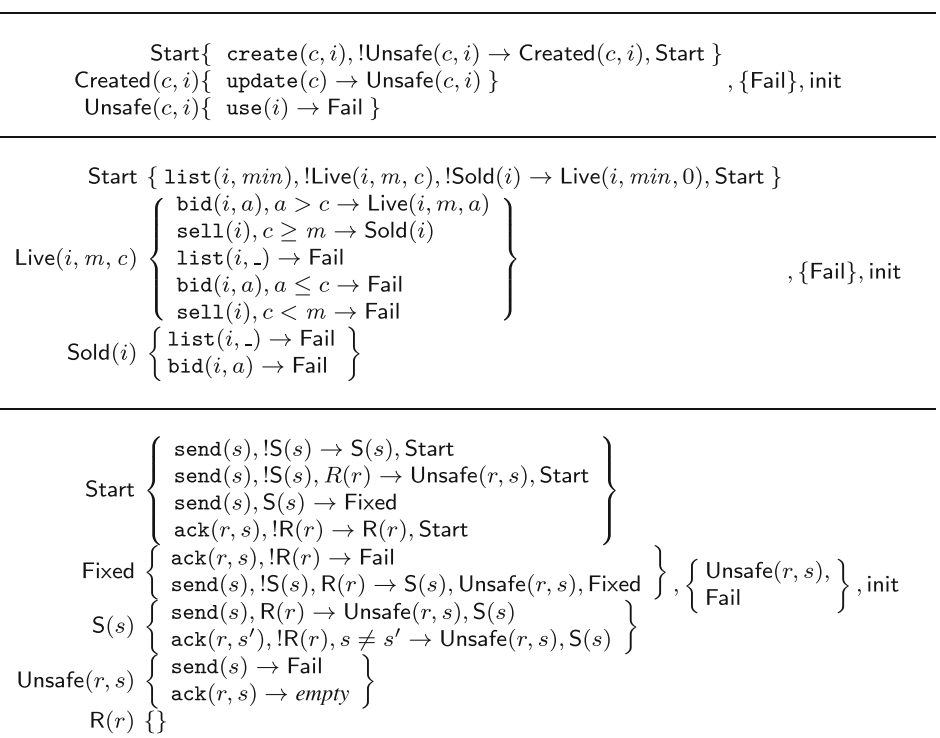


Fig. 2 Rule systems for (i) the *UnsafeIterator* property (top), (ii) the *AuctionBidding* property (middle), and (iii) the *Broadcast* property (bottom). Assuming general rule definition Fail{} and init ≡ ⟨Start, ⊥⟩

$lhs \to rhs \in \mathsf{find}(r, \theta)$. Next we define the set of ground rule expressions that result from a rule instance $\langle r, \theta \rangle$ firing in extended fact Γ as follows:

$$\mathsf{fired}(\langle r, \theta \rangle, \Gamma) = \Big\{ \theta'(rhs) \mid lhs \to rhs \in \mathsf{find}(r, \theta) \wedge \theta' = \mathsf{fire}(\Gamma, \theta, lhs) \wedge \theta' \neq \bot \Big\}$$

As $\theta'(rhs)$ is now ground we evaluate all functions to ensure that it is also pure, e.g. $[x \mapsto 1](s(x+1)) = s(1+1) = s(2)$.

Rewriting a Fact We define a rewrite relation $\Delta \xrightarrow{\mathbf{a}} \Delta'$ for facts Δ and Δ' and ground event $\mathbf{a}$ by defining the rule instances in Δ that are kept because they do not fire, are added by firing rules, and removed once fired. 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 extended fact $\Delta \cup \{\mathbf{a}\}$ and Δ_F and Δ_R are the smallest facts such that:

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

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. Note that it is important to add new rule instances *after* removing fired rule instances as a rule instance may be re-added (indeed, this is a common case).

The Property Defined by a Rule System This rewrite relation is transitively extended to traces to produce a final fact $\Delta_\tau = \mathcal{I} \xrightarrow{\tau} \Delta$, 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. Therefore, the property defined by the rule system is the set of traces τ such that $\forall \langle r, \theta \rangle \in \Delta_\tau : r(\mathsf{dom}(\theta)) \notin \mathcal{B}$.

Illustrating Semantics Again, we consider whether the example trace τ from Sect. 4 is accepted by the *UnsafeIterator* rule system given in Fig. 2(i).

We begin with the initial fact $\{\langle \mathsf{Start}, \bot \rangle\}$ and rewrite this using the first event $\mathtt{create}(C, I1)$. The rule instance $\langle \mathsf{Start}, \bot \rangle$ fires in the extended fact $\Gamma_1 = \{\langle \mathsf{Start}, \bot \rangle, \mathtt{create}(C, I1)\}$ as $\mathsf{fire}(\Gamma_1, \bot, \mathtt{create}(c, i), !\mathsf{Unsafe(c, i)}) = [c \mapsto C, i \mapsto I1]$ by first matching the ground and non-ground events and then checking that $\mathsf{Unsafe(C, I1)} \notin \Gamma_1$. Therefore, $\Delta_F = \{\langle \mathsf{Created}, [c \mapsto C, i \mapsto I1] \rangle, \langle \mathsf{Start}, \bot \rangle\}$, which is also the resulting fact. When rewriting with $\mathtt{use}(I1)$ no rules fire and there is no work to do. Rewriting with $\mathtt{create}(C, I2)$ follows similar steps as above to give the resulting fact

$$\{\langle \mathsf{Created}, [c \mapsto C, i \mapsto I1] \rangle, \langle \mathsf{Created}, [c \mapsto C, i \mapsto I2] \rangle, \langle \mathsf{Start}, \bot \rangle\}$$

Rewriting with $\mathtt{update}(C)$ causes the first two rule instances to fire, leading to the fact

$$\{\langle \mathsf{Unsafe}, [c \mapsto C, i \mapsto I1] \rangle, \langle \mathsf{Unsafe}, [c \mapsto C, i \mapsto I2] \rangle, \langle \mathsf{Start}, \bot \rangle\}$$

Finally, rewriting with use($I2$) fires the rule instance $\langle \mathsf{Unsafe}, [c \mapsto C, i \mapsto I2]) \rangle$, adding Fail to the fact. As the final fact contains a rule instance using a bad rule expressions, this trace is not in the property of the *UnsafeIterator* rule system.

Comparison with other rule-based approaches We briefly highlight how our presentation here differs from the presentation of RULER and compare it to other rule-based specification languages for runtime verification. Firstly, it is worth noting that the original papers on RULER [3,7] did not give an explicit definition of the structure and semantics of *parametric* rule systems (those using data). Instead it introduced a propositional language and described extensions to handle parameters. A restricted version of the language was described in more detail as RULER- LITE [17] which shares some similarity to our presentation here but uses rule *modifiers* (which we leave as syntactic sugar) and, crucially, doesn't provide semantics. The syntactic sugar of rule modifiers is the main difference between the rule system presented here and that used in the RULER tool. It is a similar case with the TRACECONTRACT [6], and LOGFIRE [20] tools that followed RULER—their logics can be viewed as extensions of the system presented here. Therefore, the translation described next could be straightforwardly modified to target these alternative rule-based systems.

7 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). The SCALA implementation is very close to the pseudocode presented in this section.

7.1 An equivalent representation with labelled states

We introduce an annotation of QEA that replaces states with so-called *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, thus making explicit the variables necessarily bound at a 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.

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 Fig. 3 (top). This requires introducing three new states, whereas the *UnsafeIter* QEA is already well labelled (see Fig. 4).

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 $e(\overline{x})$ 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. The translation is captured in Algorithm 1 which relies on an implicit simplify function that simplifies guards to detect a *false* guard. The general structure of the algorithm is a standard breadth-first exploration of the generated state space. Note that final states must have the full 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.

7.2 A domain-explicit form

We make the following observation about the *Broadcast* property. Consider the trace send(1).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, 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 $(\langle 2, \{s\}\rangle, \mathtt{ack}(r, x), x \neq s, \langle 2, \{r, s\}\rangle)$, which is one of two transitions added by the following construction as illustrated in Fig. 3 (bottom). However, in general, we may need to add many similar transitions to capture all possible valuation

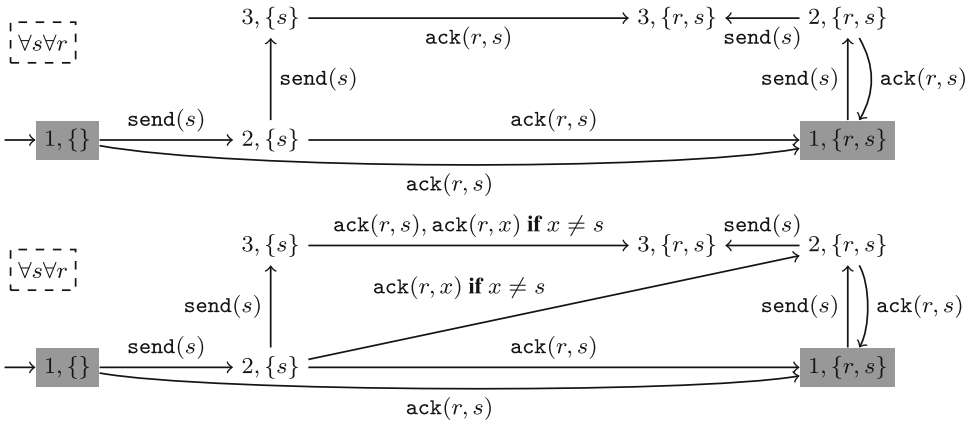


Fig. 3 *Well-labelled* and *domain-explicit* versions of the *Broadcast* QEA

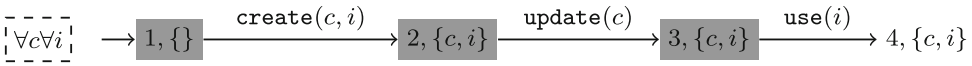


Fig. 4 A *well-labelled* version of the *UnsafeIterator* QEA

Algorithm 1: Algorithm for the translation to well-labelled form.

```
Function well-labelled is
    input  : QEA ⟨X, Q, A(X ∪ Y), δ, F, q0, σ0⟩
    output: Well-labelled equivalent QEA
    start = ⟨q0, ∅⟩;
    states = ∅; final = ∅; transitions = ∅;
    todo = Empty Queue; todo.enqueue(start);
    while todo not empty do
        ⟨state,vars ⟩ := todo.dequeue();
        for (q, e(x̄), α, γ, q′) ∈ δ where q =state do
            newstate = ⟨q′, vars + x̄/Y⟩; addstate(newstate);
            transitions.add((⟨state,vars⟩, e(x̄), α, γ,newstate));
        end
        for e(x̄) ∈ A(X ∪ Y) do
            joint_guard := true;
            for (q, e′(x̄′), γ, α, q′) ∈ δ where q =state and
            e(x̄) = e′(x̄′) do
                joint_guard := simplify(joint_guard ∧¬γ);
                if joint_guard = false then Skip to next e(x̄);
            end
            newstate = ⟨state, vars + x̄/Y⟩; addstate(newstate);
            transitions.add((⟨state,vars⟩, e(x̄),g,identity,newstate));
        end
    end
    return ⟨X,states, A(X ∪ Y),transitions, final, start, σ0⟩
end
Function addsate(newstate) is
    if newstate ∉ states then
        states.add(newstate); todo.enqueue(newstate);
        if state ∈ F and (vars ∪(x̄/Y)) = X then
        final.insert(newstate); ;
    end
end
```

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 $\overline{x} \cap (X/S) \neq \emptyset$ (e.g. the event contains at least one quantified variable not in S) we add a set of transitions

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

where R is a non-empty subset of $S \cap (\overline{x}/Y)$ (i.e. some quantified variables already bound in the state and also used in the event) and $\mathsf{fresh}(x)$ produces a consistent fresh-variable (e.g. it always maps x to the same fresh-variable). These new 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 $\in(\overline{x}[x_i \mapsto \mathsf{fresh}(x_i) \mid x_i \in R]) = \in(\overline{x})$ as $S \cap (\overline{x}/Y) = \emptyset$. Otherwise, a new event is created replacing one or more known quantified variables (in S) by fresh unquantified variables along with a guard saying that these cannot take the same value as their quantified version. This translation step is captured in Algorithm 2. Note that only new transitions are added. In particular, the set of final states must remain the same as any new states, by construction, cannot be labelled with X.

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.

The domain-explicit version of the *UnsafeIterator* QEA is given in Fig. 7 (on page 24). This is significantly more complicated due to the need to introduce many new transitions (and states) to capture the domain.

Algorithm 2: Algorithm for the translation to domain-explicit form.

Function *domain-explicit* **is**
  input : Well-labelled QEA $\langle X, Q, \mathcal{A}(X \cup Y), \delta, \mathcal{F}, q_0, \sigma_0\rangle$
  output: Domain-explicit well-labelled equivalent QEA
  transitions = δ;
  for $\langle q, S\rangle \in Q, e(\overline{x}) \in \mathcal{A}(X \cup Y)$ *where* $(\overline{x} \cap (X/S)) \neq \emptyset$ **do**
    foreach *subset R of* $S \cap (\overline{x}/Y)$ *where* $R \neq \emptyset$ **do**
      $\overline{y}$:= []; guard := true;
      for $v \in \overline{x}$ **do**
        if *v is a variable* **then** w := `fresh`(v); $\overline{y}$ .append(w); guard := guard $\wedge v \neq w$;
        else $\overline{y}$ .append(v);
      end
      transitions.add(($\langle q, S\rangle, e(\overline{y})$,guard,identity,$\langle q, S \cup (\overline{x}/R)\rangle$));
    end
  end
  return $\langle X, Q, \mathcal{A}(X \cup Y)$,*transitions*, $\mathcal{F}, q_0, \sigma_0\rangle$
end

7.3 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 $\langle\langle 2, \{i\}\rangle, \texttt{bid}(i, a)$, **if** $a > c, c := a, \langle 2, \{i\}\rangle\rangle$ from the labelled QEA for the *AuctionBidding* property (see Fig. 5). 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:

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

This is problematic as $\texttt{bid}(i, a)$ 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 transition would become $\langle\langle 2, \{i\}\rangle, \texttt{bid}(i, b)$, **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:

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

The resultant QEA is clearly equivalent as all paths remain the same. The implementation of this translation is trivial and we assume a function *fresh-variable* that runs on a domain-explicit, well-labelled QEA.

7.4 The translation

Given a fresh-variable 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

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

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, Fig. 6 gives the set of rule definitions generated by our tool for (i) the domain-explicit labelled QEA for the *Broadcast* property, and (ii) the domain-explicit labelled QEA in fresh-variable form for the *AuctionBidding*.

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 = $\{\langle r_{q_0}, \sigma_0\rangle\}$.

Algorithm 3 captures this translation. As expected, there is some complexity in how we handle Y (either renaming these to fresh-variables or applying α or σ_0). As before,

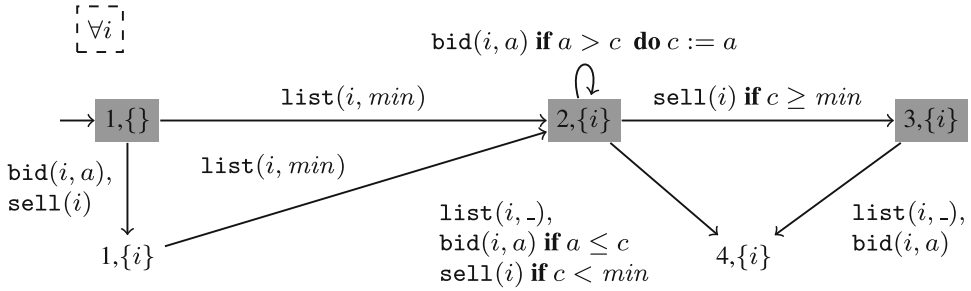


Fig. 5 A *well-labelled domain explicit* version of the *AuctionBidding* QEA

Fig. 6 Resulting rule systems for (i) *Broadcast property* (top), and (ii) *AuctionBidding* property (bottom)

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

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

we implicitly move between sets and lists, assuming an implicit ordering on elements to ensure turning a set into a list produces a unique list. The overall translation process can be achieved by chaining the four algorithms *well-labelled*, *domain-explicit*, *fresh-variable*, and *translation*.

The translation is *decidable*; any QEA of the form given in Sect. 5 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.

8 Correctness of the translation

Here we show that the translation of Sect. 7 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* **a** *be a ground event. The rule instance $\langle r_i, \theta_i \rangle \in \Delta$ will fire for* **a** *with a rule term of kind (ii) to produce $\langle r_k, \theta_k \rangle$ if and only if $\theta_k \in$* extensions$(\theta_i, \mathbf{a})$ *for the domain-explicit labelled form of the* QEA.

Proof We note that this is the exact property that the domain-explicit form is introduced to achieve. Firstly, for the domain-explicit form 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

Algorithm 3: Algorithm for the translation to rule system.

```
Function translate is
    input : Fresh-variable, domain-explicit, well-labelled QEA
            ⟨X, Q, A(X ∪ Y), δ, F, q0, σ0⟩
    output: Rule System RS
    definitions = ∅;
    for ⟨state, vars⟩ ∈ Q do
        parameters := vars ++ Y; body := [];
        for (⟨q, S⟩, e(x̄), α, γ, ⟨q′, S′⟩) ∈ δ where q =state and
        S =vars do
            // All rule terms start with event
               and guard
            left := [e(x̄), γ];
            // Case (i) transitions with the
               same label
            if S = S′ then right := [r_q′(S, α(Y))] ;
            // Case (ii) transitions extending
               the label
            else
                negated_rules = [];
                for ⟨q1, S1⟩ ∈ Q where vars ⊂ S1 do
                    Y' = []; foreach y ∈ Y do Y'.append(y_(q1,S1));
                    negated_rules.append(!r_q1(S1++Y′));
                end
                left := left ++ negated_rules;
                Y' = []; foreach y ∈ Y do Y'.append(α(y)) ;
                right := [r_q′(S′, Y′), r_q(S, Y)]
            end
            body.append(left ⇒ right);
        end
        definitions.add((r_q, parameters, body));
    end
    Y' := []; foreach y ∈ Y do
        Y'.append(σ0(y))
    end
    initial := {r_q0(Y′)};
    bad := ∅;
    for ⟨q, S⟩ ∈ Q where ⟨q, S⟩ ∉ F and S = X do
    bad.add(r_q(S++Y)); ;
    return (definitions,bad,initial)
end
```

to ensure there will be an event which, when matched with, will produce this submap.

As defined on page 8, the set $\mathsf{extensions}(\theta_i, \mathbf{a})$ is given by extending θ_i by consistent non-empty 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 **b** that could extend the label. Therefore, due to how the rule system is constructed, there will be a rule term of kind (ii) for every such event in the body of the associated rule definition. In more detail, if we take the *if* direction, for $\theta_k \in \mathsf{extensions}(\theta_i, \mathbf{a})$ we need an event in the (original) alphabet matching the ground event and the some part of the resulting valuation extending an existing valuation in a consistent way. As mentioned above, the sub-maps are captured explicitly by an extra event in the alphabet. Hence, it is guaranteed that such an event exists and a transition exists for the event. For the *only if* direction note that the translation only adds rule terms for transitions in the automaton. Note that the two systems check the predicate γ at different points. This is why the domain-explicit form needed to ensure that all possible valuations were covered by some transition, so that the above statement about a transition always existing is true. □

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 1 *Given a domain-explicit labelled* QEA *with $Y = \emptyset$, let RS be the rule system given by the above translation. For monitoring state M_τ and fact Δ_τ if*

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

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 **a** are the same.

Let us take (i) first. According to Definition 4 if **a** is not relevant then no rules are fired, which matches with no transitions being taken in the single step construction. If **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. □

Finally, we need to consider the case of a QEA with non-empty Y. Firstly, Lemmas 1 and 2 can be lifted to this case by explicitly identifying and excluding variables in Y in valuations and rule definitions. Note that these lemmas are solely about relating the domain of the monitoring state to the quantified parts of rule instances and are not affected by this part of the valuation. Therefore, we will use these lemmas for QEA with non-empty Y in our following proof.

Theorem 2 *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*

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

then for any valuation θ

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

Proof (*Sketch*) We need to make certain adjustments to the proof of Theorem 1 to convince ourselves that this part of the valuation is correctly treated. The structure of the argument is the same. In part (i), we must consider each $\langle q, \sigma\rangle \in M_{\tau'}(\theta)$ and corresponding $\langle r_q, \theta \dagger \sigma\rangle \in \Delta$. The only new behaviour to consider in the firing of the rule terms of $r_q(\mathsf{dom}(\theta))$ is the recording of unquantified variables and the application of guards and assignments. Here the same guards and assignments are used, which will lead to the same configurations/rule instances being blocked and updated. Part (ii) can be updated to use versions of Lemmas 1 and 2 that refer to configurations rather than states—the presence of unquantified variables has no impact on these. □

9 A general notion of redundancy inspired by the translation

Consider the rule system given in Fig. 7 resulting from the translation of the *UnsafeIterator* 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 not be surprising as if we remove these rule definitions the rule system becomes very similar to the one given in Sect. 4, 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. We now formalise this notion of redundancy and then show how it can be used to optimise the translation.

9.1 Demonstrating redundancy

Before we give the formal definition of redundancy, let us explore the notion of redundancy using an extension of the *UnsafeIterator* property[3]. The QEA in Fig. 8 captures the property that iterators for collections created from a map (e.g. the keySet) should not be used after the original map is updated. Consider the trace

$$\texttt{create}(A, X).\texttt{iterator}(X, 1).\texttt{use}(1).\texttt{create}(B, Y).\ \texttt{iterator}(Y, 2).\texttt{use}(2)$$

The only valuations of interest are $[m \mapsto A, c \mapsto X]$, $[m \mapsto A, c \mapsto X, i \mapsto 1]$, $[m \mapsto B, c \mapsto Y]$, and $[m \mapsto B, c \mapsto Y, i \mapsto 2]$. Let us consider why two other valuations are redundant. Firstly, consider $[i \mapsto 1]$. This has a non-empty projection for the trace (use(1)) but that projection remains in the initial state—so does not provide any new information. Secondly, consider $[m \mapsto A, c \mapsto X, i \mapsto 2]$ as an extension of $[m \mapsto A, c \mapsto X]$. Again this has a non-empty trace projection which is the same as that for the valuation it extends and no new information is recorded.

The observation we make is that if a valuation extends a smaller valuation without adding any new information (changing the set of configurations mapped to by the valuation) it is redundant and can be omitted. There is a caveat to this—if the the smaller valuation contains information that would affect the verdict this should be preserved and the larger valuation created.

[3] We introduce this new property as it makes a special case of redundancy more obvious. In *UnsafeIterator* we only make redundant extensions of the initial configuration, but in this new property we also make redundant extensions of intermediate configurations.

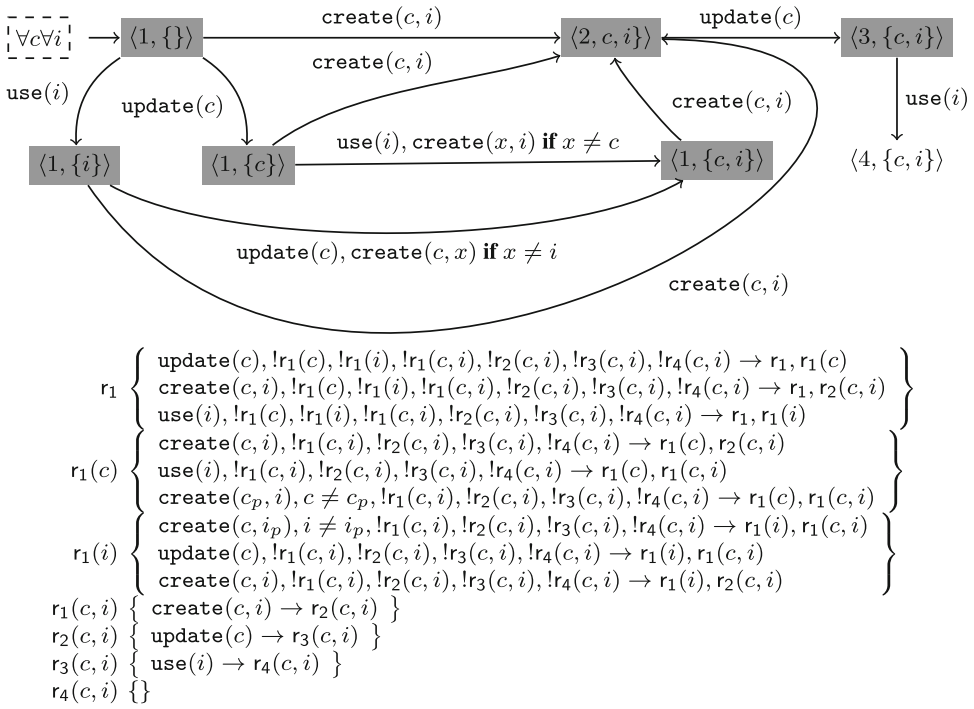


Fig. 7 Fully transformed QEA and corresponding rule system for the *UnsafeIterator* property

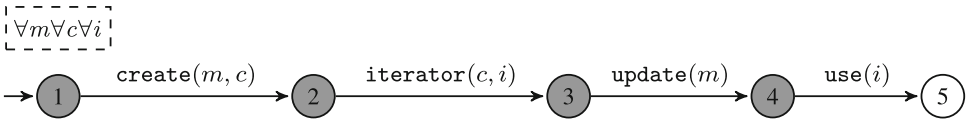
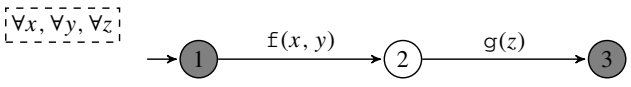


Fig. 8 A QEA for the UnsafeMapIterator property for Java

9.2 Defining redundancy

The above discussion hinges on the notion that valuations mapping to empty or trivial projections can be ignored. The empty projection would only affect the verdict if the initial state were non-final (this is because all quantification is universal, if existential quantification were allowed this would change). We call a QEA *normal* if its initial state is final. All QEAs that the authors have written or encountered have been normal. The empty projection is a special case of a more general case for trivial projections. In the above example all trivial projections (and hence redundant valuations) were in a final state, e.g. they would not violate the property (assuming a normal QEA). This is a necessary condition (for a valuation to be redundant) but not sufficient. As an example, consider the following QEA



and the trace $\mathtt{g}(1).\mathtt{f}(2,3)$. This clearly violates the property as the trace reaches non-final state 2 for the valuation $[x \mapsto 2, y \mapsto 3, z \mapsto 1]$. However, after the first event we have the valuation $[z \mapsto 1]$ in the final initial state. If we were to remove this valuation, then we would fail to produce the total valuation $[x \mapsto 2, y \mapsto 3, z \mapsto 1]$. The reason for this is that we can get to a non-final state without rebinding z.

We will say that a valuation (in the domain of the monitoring state) contains necessary information and should be preserved if it is *active*. A valuation is active if it may not be reconstructed before reaching a non-final active state (which subsumes the case where it is already in a non-final state). This may happen exactly when there exists a path from the current state to a non-final state that does not bind the current valuation's quantified variables.

Definition 6 (*Active Valuation*) Given a normal QEA $\langle X, Q, \mathcal{A}(X \cup Y), \delta, \mathcal{F}, q_0, \sigma_0 \rangle$ and a monitoring state M, a valuation $\theta \in \mathsf{dom}(M)$ is *active* if there is a configuration $\langle q, \sigma \rangle \in M(\theta)$ such that $\mathsf{dom}(\theta) \not\subseteq \mathsf{bpath}(q, \emptyset)$ where $\mathsf{bpath}(q, S)$ is defined recursively as

$$\begin{array}{ll} \mathsf{bpath}(q, S) = S \cap X & \text{if } q \notin \mathcal{F} \\ \mathsf{bpath}(q, S) = X & \text{if } \{(q, \mathbf{b}, \gamma, \alpha, q') \in \delta\} = \emptyset \\ \mathsf{bpath}(q, S) = \bigcap_{(q, e(\overline{z}), \gamma, \alpha, q') \in \delta} \mathsf{bpath}(q', S \cup \overline{z}) & \text{otherwise} \end{array}$$

The bpath function builds the set of variables used on every path to a non-final state, hence taking the intersection of the variables on each path. To ignore paths ending in final states, we associate such paths with the maximum set of quantified variables X. Note that if $q \notin \mathcal{F}$ then $\mathsf{bpath}(q, \emptyset) = \emptyset$

and $\mathsf{dom}(\theta) \not\subseteq \emptyset$ unless $\theta = \bot$, in which case θ cannot be redundant.

Note that this is a very strong requirement and that being inactive does not mean that a valuation is not needed but that it is needed even if a smaller valuation exists in the same state. A valuation is then redundant if it will always be recreated before it is needed, e.g. it is not active and it extends another valuation in the same configurations.

Definition 7 (*Redundant Valuation*) Given the monitoring state M, let $\theta_L, \theta \in \mathsf{dom}(M)$ be two valuations such that θ_L is the largest (wrt $\sqsubseteq$) valuation in $\mathsf{dom}(M)$ such that $\theta_L \sqsubset \theta$. The valuation θ is *redundant with respect to* θ_L iff it is not active and $M(\theta) = M(\theta_L)$, in which case we write $\mathsf{redundant}(\theta, \theta_L, M)$.

The monitoring construction of Definition 4 can be updated to use this form of redundancy such that at any step the monitoring state does not contain any redundant valuations.

Definition 8 (*Monitoring Construction up to Redundancy*) We extend the construction of monitoring state $(\mathbf{a} * M)$ given in Definition 4 as follows. Let $(\mathbf{a} * M) = K_n$ where $K_0 = \bot$ and

$$K_i = \left[(\theta_j \mapsto C_j) \in N_i \mid \begin{array}{l} \theta_j \in \mathsf{dom}(K_{i-1}) \vee (\theta_j \neq \theta_i \wedge \neg \mathsf{redundant}(\theta_j, \theta_i, N_i)) \vee \\ (\theta_j = \theta_i \wedge \nexists \theta' \in \mathsf{dom}(K_{i-1}).\mathsf{redundant}(\theta_i, \theta', N_i)) \end{array} \right]$$

In other words, an entry in N_i is kept if it was already kept at the last iteration or if it is not redundant. We capture two cases for redundancy. Either the valuation was freshly introduced, in which case we need to check redundancy with respect to θ_i or it was an existing valuation, in which case we need to check whether it should be removed. We separate the two cases to (i) show that there is a reasonable way to compute such redundant valuations, and (ii) highlight the fact that we may only wish to focus on one kind of redundancy, e.g. preventing adding redundant valuations vs explicitly removing them.

Theorem 3 *Removing redundant bindings from a monitoring state preserves the set of accepted traces.*

Proof (*Sketch*) Let us assume that it does not. This means that there is either (i) a total valuation missing from the final monitoring state that should be in a non-final state, or (ii) a total valuation in the final monitoring state is wrongly associated with a non-final state, which can only be due to the valuation being created from the wrong set of configurations initially. For either case to occur, the valuation needed to create the total valuation would need to be absent from the monitoring state or in the wrong configuration. However, if it has been removed and is not present when needed then it would have been active or map to distinct configurations, and thus not removed. □

9.3 Optimising the translation

We can use this notion of redundancy to optimise the translation from QEA to rule systems as some transitions in a well-labelled domain-explicit QEA will only ever be taken by projections associated with redundant valuations.

This can be achieved by a final transformation. Simply, a labelled state $\langle q, S \rangle$ is redundant if it is final and there exists another labelled state $\langle q, S' \rangle$ such that $S' \subset S$ and all outgoing transitions of $\langle q, S \rangle$ are included in the outgoing transitions of $\langle q, S' \rangle$. In such a case it is guaranteed that staying in $\langle q, S' \rangle$ instead of transferring to $\langle q, S \rangle$ will lead to the same total valuations in the same configurations.

The reason this transformation is relatively simple is that the reachable quantified variables have already been made explicit, e.g. we do not need the complicated bpath computation in the definition of active valuation above.

The well-labelled domain-explicit QEA for the *UnsafeIterator* property in Fig. 7 can be transformed by first removing state $\langle 1, \{c, i\} \rangle$ and then the states $\langle 1, \{i\} \rangle$ and $\langle 1, \{c\} \rangle$. The result of this transformation and the final rule system are given in Fig. 9. The final rule system is very similar to that given in Fig. 2. The only difference is the additional check $!r_2(c, i)$. However, we can independently identify this check as redundant as the rule introduces $r_2(c, i)$.

Evaluating the Optimisation As a small demonstration of the value of this optimisation on the translation itself we perform a small experiment with the *UnsafeIterator* property. We produce four representative traces of different lengths by varying the number of collections, iterators, uses, and updates. We then compare the performance of monitoring those traces using three different rule systems—the original hand-crafted rule system, the rule system resulting from the translation, and the rule system resulting from the optimised translation. The monitoring algorithm used is an implementation of the definitions presented in this paper rather than the original RULER system. This algorithm and the code required to reproduce the experiment are available in the GitHub repository.

Table 1 gives the results. The numbers are an average of three runs (after an initial warm-up of ten runs to account for JIT-effects). Here we can see that the performance for the original and optimised rule systems is very similar, whilst

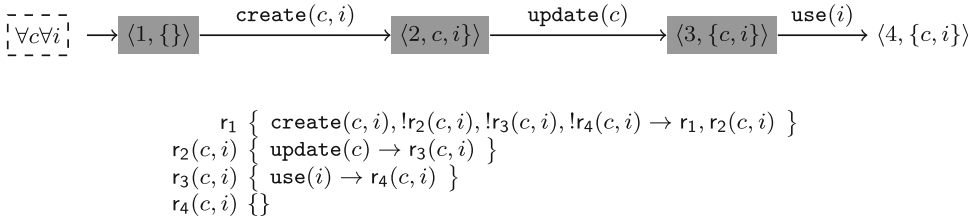


Fig. 9 Optimised transformation and updated rule system for *UnsafeIterator* property

Table 1 Running times (in seconds) monitoring various traces for the *UnsafeIterator* property using the original, translated, and optimised rule systems

Events	Original	Translated	Optimised
2246	0.025	0.08	0.02
12,353	0.59	1.86	0.54
23,583	2.16	7.41	2.21
113,333	55.6	175.4	53.06

the original translated rule system performs much worse (roughly 3x worse). In fact, the optimised rule system sometimes performed marginally better than the original one. Times for the translated and optimised version do not include the time for translation but this is negligible.

9.4 A general notion of redundancy

Here we briefly show that the above notion of redundancy is *general* as it can be used to explain existing techniques for blocking or removing valuations.

9.4.1 Blocking redundant valuations

Firstly, we consider cases where we block the introduction of a redundant valuation.

Creation events in JavaMOP are used to explicitly identify events, such that a valuation is only created (as an extension of the empty valuation) for a creation event. Creation events not belonging to the following set can violate the monitoring semantics:

$$\mathsf{creation} = \{\mathbf{a} \in \mathcal{A} \mid \mathsf{fvars}(\mathbf{a}) \neq \emptyset \vee \neg\exists(q_0, \mathbf{b}, \_, \_, \_) \in \delta : \mathsf{qvars}(\mathbf{a}) \subset \mathsf{qvars}(\mathbf{b})\}$$

where fvars selects the free variables (in Y) in an event and qvars selects the quantified variables (in X) in an event. This ensures that we only create valuations that are not associated with a trivial configuration. The check for events labelling transitions out of q_0 with strictly more quantified variables ensures that the the produced valuation would not be active.

Enable sets were introduced in JavaMOP [29] to reduce work. In JavaMOP there is a notion of a *goal* verdict and the idea is to exclude trace projections that cannot reach this verdict. This is done by identifying events that must occur before other events in a valid trace—if they have not occurred then the projection will not be valid. To improve efficiency, the set of parameters that must be bound (called the enable set) is used instead of events. This is a special instance of our redundancy as a trace not able to reach a goal verdict would be in an inactive state; however, this only applies to cases where it is also the case that no active states are reachable.

9.4.2 Removing redundant valuations

Next we consider cases where we can remove valuations that become redundant. Valuations can become redundant if we know that certain events cannot occur in the future as the objects they refer to no longer exist in the monitored system, i.e. become garbage. Let us revisit the UnsafeMapIter example (Fig. 8). Consider the valuation $[m \mapsto A, c \mapsto X, i \mapsto 1]$. If iterator 1 becomes garbage at any point, there is no way to reach state 5, and therefore, this valuation can no longer influence the verdict and can be safely removed. However, we cannot remove the valuation $[m \mapsto A, c \mapsto X]$ when collection X becomes garbage if it has reached state 3, as it is still possible to reach state 5 without any events involving c. However, if we were in state 2 we could remove the valuation. A valuation can be safely removed if an active state is not reachable using only those events that involve quantified variables whose values have not become garbage. In the case of partial valuations, we must consider whether the valuation could be extended to reach an active state. Similar techniques have been used previously. An approach similar to our own is taken in [2] where variables are categorised based on whether a final state is reachable. In [26], "coenable sets" are developed as a dual to enable sets. These identify the parameters that must be bound to reach a goal verdict from any state.

9.4.3 Discussion

Two obvious questions at this point are (i) is this notion of redundancy helpful and (ii) does this general notion of redundancy *add* anything to existing work. We address these two questions here.

Firstly, we should note that the existing redundancy elimination techniques outlined in this section are *necessary* for

the performance of the JavaMOP and MarQ tools. In general, the complexity of the monitoring algorithms of both tools is dependent on the number of valuations of quantified variables constructed and kept. Therefore, any method that reduces this number has a direct impact on performance, especially as this number will generally be the cross-product of the domains of quantified variables. There are various papers on JavaMOP that document precisely the performance benefits of these ideas. For example, they show [13] that the enable set optimisation can reduce monitoring overhead by about 20% in common cases and can make the monitoring problem tractable in others. Further, Dongyun Jin showed in his thesis [25] that the combination of these optimisations would often half the monitoring overhead. There are no similar published results for MarQ, but the general result is the same.

Secondly, the main contribution here is a general formalisation of redundancy criteria where existing optimisations can be seen as specific instances. So far, this has not led to new optimisations being identified but has simplified their implementation in MarQ and paves the way to explore more refined redundancy elimination techniques—some compatible ideas have already been discussed in theory [31].

10 Discussion and related work

We explore what we have learned about the relationship between the two languages introduced in Sects. 5 and 6 by the development of the previous translation. We consider the *expressiveness* of the languages, the *efficiency* of monitoring, how data are 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 [22] 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 [4].

- *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 Fig. 10. 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 [22], but this relies on explicitly recording all facts and performing a computation on a special end of trace event.
- *Non-Determinism.* In [4], QEA were given some-path non-determinism, but in [22] 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 [32] 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). To be clear—we could extend our notion of a rule system and our translation to support non-determinism and the extension would be relatively straightforward but add an extra layer of complexity.

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 [8], to rule systems, which do not [21]. This appears to be the wrong direction to make gains in efficiency. However, we have already used the translation to identify a general notion of redundancy (introduced in Sect. 9) which can be used to significantly improve the performance of the monitoring algorithm for QEA. In the other direction, 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. Recall that after removing the redundancy, the first rule definition in the translation of the *UnsafeIterator* QEA becomes

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

which, when compared to the rule system in Fig. 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

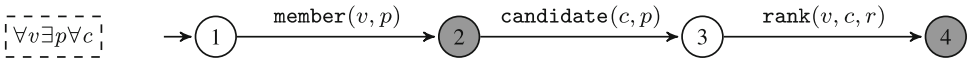


Fig. 10 A QEA for the *CandidateSelection* property taken from [4]

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 [29] has made the slicing approach popular with its efficient implementation. The QEA formalism [4,30] 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 [3,5] and are used in TraceContract [6] and LOGFIRE [21] 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.

11 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. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. SIGPLAN Not. **40**, 345–364 (2005)
2. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitors feasible. SIGPLAN Not. **42**(10), 589–608 (2007)
3. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for runtime monitoring: from EAGLE to RuleR. J. Logic Comput. **20**(3), 675–706 (2010)
4. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: towards expressive and efficient runtime monitors. In: FM, pp. 68–84 (2012)
5. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: VMCAI, pp. 44–57 (2004)
6. Barringer, H., Havelund, K.: Tracecontract: a Scala DSL for trace analysis. In: Proceedings of the 17th International Conference on Formal Methods, pp. 57–72. Berlin, Heidelberg (2011)
7. Barringer, H., Havelund, K., Rydeheard, D., Groce, A.: Rule systems for runtime verification: a short tutorial. In: Bensalem, S., Peled, D.A. (eds.) Runtime Verification, pp. 1–24. Springer, Berlin (2009)
8. Bartocci, E., Bonakdarpour, B., Falcone, Y., Colombo, C., Decker, N., Klaedtke, F., Havelund, K., Joshi, Y., Milewicz, R., Reger, G.,

Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification. In: International Journal on Software Tools for Technology Transfer (STTT) (2017)
9. Bartocci, E., Falcone, Y., Francalanza, A., Leucker, M., Reger, G.: An introduction to runtime verification. In: Lectures on Runtime Verification—Introductory and Advanced Topics, volume 10457 of LNCS. Springer, pp. 1–23 (2018)
10. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: Monpoly: monitoring usage-control policies. In: Khurshid, S., Sen, K. (eds.) Runtime Verification. Lecture Notes in Computer Science, vol. 7186, pp. 360–364. Springer, Berlin (2012)
11. Bauer, A., Küster, J.-C., Vegliach, G.: The ins and outs of first-order runtime verification. Formal Methods in System Design, pp. 1–31 (2015)
12. Bozzelli, L., Sánchez, C.: Foundations of boolean stream runtime verification. Theor. Comput. Sci. **631**, 118–138 (2016)
13. Chen, F., Meredith, P.O., Jin, D., Rosu, G.: Efficient formalism-independent monitoring of parametric properties. In: 2009 IEEE/ACM International Conference on Automated Software Engineering IEEE, pp. 383–394 (2009)
14. Chen, F., Roşu, G.: 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, pp. 246–261 (2009)
15. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: Proceedings of the 12th International Symposium on Temporal Representation and Reasoning, pp. 166–174 (2005)
16. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. In: Tools and Algorithms for the Construction and Analysis of Systems—20th International Conference, TACAS 2014, pp. 341–356 (2014)
17. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D. (eds.) Summer School Marktoberdorf 2012—Engineering Dependable Software Systems. IOS Press, Amsterdam (2013)
18. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: Proceedings of the 18th International Conference on Runtime Verification, pp. 241–262 (2018)
19. Hallé, S., Khoury, R.: 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, pp. 251–258 (2015)
20. Havelund, K.: Rule-based runtime verification revisited. Int. J. Softw. Tools Technol. Transf. **17**(2), 143–170 (2015)
21. Havelund, K.: Rule-based runtime verification revisited. Int. J. Softw. Tools Technol. Transf. **17**(2), 143–170 (2015)
22. Havelund, K., Reger, G.: 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, pp. 151–189 (2015)
23. Havelund, K., Reger, G.: Runtime verification logics—a language design perspective. In: KIMfest 2017. Springer (2017)
24. Havelund, K., Reger, G., Zalinescu, E., Thoma, D.: Monitoring events that carry data. In: Lectures on Runtime Verification—Introductory and Advanced Topics, volume 10457 of LNCS. Springer, pp. 60–97 (2018)
25. Jin, D.: Making Runtime Monitoring of Parametric Properties Practical. Ph.D. thesis, University of Illinois at Urbana-Champaign, August (2012)
26. Jin, D., Meredith, P.O.N., Griffith, D., Rosu, G.: Garbage collection for monitoring parametric properties. SIGPLAN Not. **46**(6), 415–424 (2011)
27. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebraic Program. **78**(5), 293–303 (2008)
28. Medhat, R., Joshi, Y., Bonakdarpour, B., Fischmeister, S.: Parallelized runtime verification of first-order LTL specifications. Technical report, University of Waterloo (2014)
29. Meredith, P., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. J. Softw. Tools Technol. Transf. 1–41 (2011)
30. Reger, G.: Automata Based Monitoring and Mining of Execution Traces. Ph.D. thesis, University of Manchester (2014)
31. Reger, G.: A story of parametric trace slicing, garbage and static analysis. Electron. Proc. Theor. Comput. Sci. **254**, 1–14 (2017)
32. Reger, G., Cruz, H.C., Rydeheard, D.: 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)
33. Reger, G., Hallé, S., Falcone, Y.: Third international competition on runtime verification CRV 2016. In: RV 2016 (2016)
34. Reger, G., Rydeheard, D.: From first-order temporal logic to parametric trace slicing. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22–25, 2015. Proceedings, pp. 216–232. Springer International Publishing, Cham (2015)
35. Reger, G., Rydeheard, D.: From parametric trace slicing to rule systems. In: International Conference on Runtime Verification. Springer, pp. 334–352 (2018)
36. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. In: Proceedings of the 5th International Workshop on Runtime Verification (RV'05), volume 144(4) of ENTCS. Elsevier, pp. 109–124 (2006)