From Parametric Trace Slicing to Rule Systems

Giles Reger David Rydeheard

University of Manchester, Manchester, UK

November 11, 2018

Spot the Difference

```
qea(UnsafeMapIterator) {
  forall(m,c,i)
  accept skip(start){
    create(m, c) →
          createdC
  }
  accept skip(createdC){
    iterator(c, i) →
          createdI
  }
  accept skip(createdI){
    update(m) → updated
  }
  accept skip(updated){
    next(i) → failure
  }
}
```

$$\square \forall i. \Big(\mathtt{next}(i) \rightarrow \exists m, c. \big(\neg \mathtt{update}(m)\, \mathsf{S}\, (\mathtt{iterator}(c, i) \wedge \blacklozenge\, \mathtt{create}(m, c))\big)\Big)$$

$$\begin{aligned}
\mathtt{createdC} &= \mathtt{create} \cup \mathtt{createdC}[-1|\emptyset] \\
\mathtt{createdI} &= (\mathtt{iterator} \bowtie \mathtt{createdC}) \cup \mathtt{createdI}[-1|\emptyset] \\
\mathtt{updated} &= (\mathtt{update} \bowtie \mathtt{createdI}) \cup \mathtt{updated}[-1|\emptyset] \\
\mathtt{ok} &= \mathtt{next} \not\subseteq \pi_{\langle i \rangle}(\mathtt{updated})
\end{aligned}$$

```
class UnsafeMapIterator extends Monitor {
  val create,iterator,update,next = event
  val createdC,createdI,updated = fact

  r1: create(m,c) |-> createdC(m,c)
  r2: createdC(m,c),
      iterator(c,i) |-> createdI(m,i)
  r3: createdI(m,i),
      update(m) |-> updated(i)
  r4: updated(i), next(i) |-> error
}
```

$$\forall c, m, i.\ \square \Big(\mathtt{create}(m, c) \rightarrow \bigcirc \square(\mathtt{iterator}(c, i) \rightarrow \bigcirc \square(\mathtt{update}(m) \rightarrow \bigcirc \square\, \neg \mathtt{next}(i))) \Big)$$

Some Terminology: Parametric RV

Runtime Verification Problem

Given a trace τ and a specification φ decide whether $\tau \in \mathcal{P}(\varphi)$.

Where a trace is a finite sequence of events and $\mathcal{P}(\varphi)$ is the set of traces denoted by specification φ (could be good prefixes).

This can be parameterised by our notion of event.

Propositional Events

An event is an atomic symbol

Parametric Events

An event is a pair of an event name and a finite sequence of values

We Have Lots of Languages for Parametric RV

Rule-based

- RuleR, LogFire, TraceContract

Temporal logic

- 'Standard' First-order LTL (past/future)
- MFOTL (also with aggregates),
- Parameterized LTL, LTL-FO$^+$, LTL$^{\text{FO}}$, LTL-FO
- LTL and MTL extended with freeze quantifiers

Projection or Slicing based

- JavaMOP, tracematches, QEA, Larva, Mufin

Stream based

- Lola, TeSSLa, BeepBeep

Our Research Question

The Question

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.

The Approach:

1. Map the space. Find out what languages there are. Apply them to examples and see what they do (joint with Klaus).

2. Connect the space. Formalise translations from (a fragment of) one language to (a fragment of) another language.

Our Research Question

The Question

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.

The Approach:

1. Map the space. Find out what languages there are. Apply them to examples and see what they do (joint with Klaus).

2. Connect the space. Formalise translations from (a fragment of) one language to (a fragment of) another language.

Our Research Question

The Question

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.

The Approach:

1. Map the space. Find out what languages there are. Apply them to examples and see what they do (joint with Klaus).
2. Connect the space. Formalise translations from (a fragment of) one language to (a fragment of) another language.

Our Research Question

The Question

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.

The Approach:

1. Map the space. Find out what languages there are. Apply them to examples and see what they do (joint with Klaus).

2. Connect the space. Formalise translations from (a fragment of) one language to (a fragment of) another language.

Our Research Question

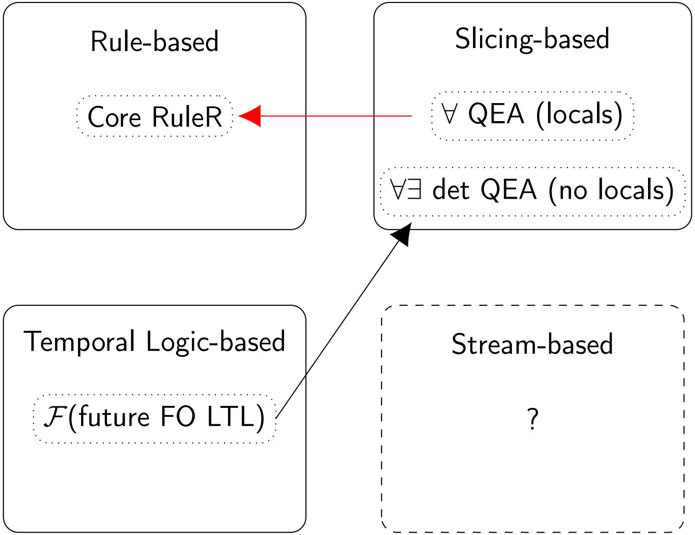
The Question

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.

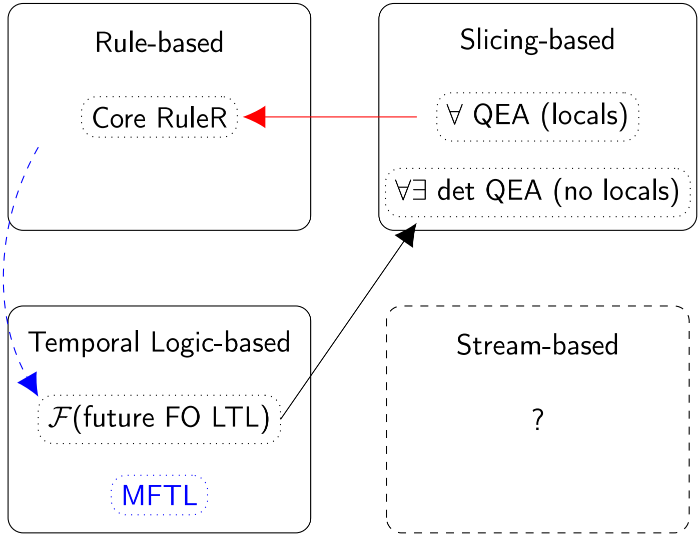
The Approach:

1. Map the space. Find out what languages there are. Apply them to examples and see what they do (joint with Klaus).
2. Connect the space. Formalise translations from (a fragment of) one language to (a fragment of) another language.

The Landscape



The Landscape



Previous Work

From First-Order Temporal Logic to Parametric Trace Slicing (RV15)

- Motivation: the semantics of the LTL plugin with parametric slicing is 'local' e.g. the notion of next is always with respect to the current slice

- Define a fragment of a function-free future-time first-order LTL that coincides with slicing-based LTL, which can be converted to a QEA

- The fragment is (in some sense) next-free, reflecting the restrictions that slicing brings

This Talk

The two languages (QEA and RuleR) by example

The translation by example

Properties of the translation

Some Observations

Fin

Two Languages By Example

Quantified Event Automata. Introduced in 2012 by Barringer, Falcone, Havelund, Reger, and Rydeheard. Based on parametric trace slicing and inspired by MOP.

RuleR. Introduced in 2007 by Barringer, Rydheard, and Havelund. Follow on work from fixed-point rule-based language EAGLE.

We will use a running example:

UnsafeIterator

An iterator object i created from a collection object c cannot be used after c is updated.

Quantified Event Automata

$$\begin{tikzcd} \boxed{\forall c \forall i} \\ {} \arrow[r] & 1 \arrow[r, "\texttt{create}(c,i)"] & 2 \arrow[r, "\texttt{update}(c)"] & 3 \arrow[r, "\texttt{use}(i)"] & 4 \end{tikzcd}$$

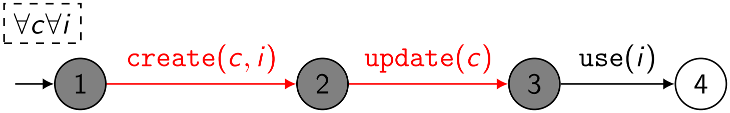
$$\texttt{create}(C,A).\texttt{create}(C,B).\texttt{use}(A).\texttt{update}(C).\texttt{use}(B)$$

Based on parametric trace slicing

- Domain of quantification extracted from trace
- For each binding of variables, slice trace and check slice
- QEA differs from MOP as it keeps the parameters in the slice

Ignore existential quantification in this work (discussed later)

Quantified Event Automata



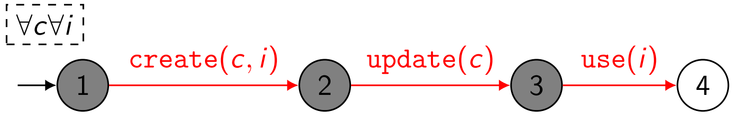
create(C, A).create(C, B).use(A).update(C).use(B)

Based on parametric trace slicing

- Domain of quantification extracted from trace
- For each binding of variables, slice trace and check slice
- QEA differs from MOP as it keeps the parameters in the slice

Ignore existential quantification in this work (discussed later)

Quantified Event Automata



$\texttt{create}(C, A).\texttt{create}(C, B).\texttt{use}(A).\texttt{update}(C).\texttt{use}(B)$

Based on parametric trace slicing

- Domain of quantification extracted from trace
- For each binding of variables, slice trace and check slice
- QEA differs from MOP as it keeps the parameters in the slice

Ignore existential quantification in this work (discussed later)

QEA Small-Step Semantics

Definition (Monitoring Construction)

Given ground event **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 $i \in [1, m]$ and $N_0 = \bot$

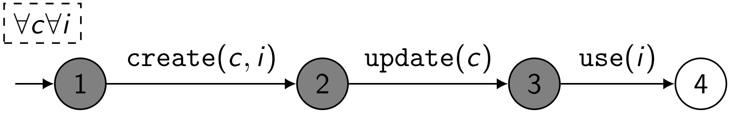
$$N_i = N_{i-1} \dagger \texttt{Add}_i \dagger \begin{cases} [\theta_i \mapsto \text{next}(M(\theta_i), \mathbf{a}, \theta_i)] & \text{if relevant}(\theta_i, \mathbf{a}) \\ [\theta_i \mapsto M(\theta_i)] & \text{otherwise} \end{cases}$$

with additions defined in terms of extensions not already present:

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

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

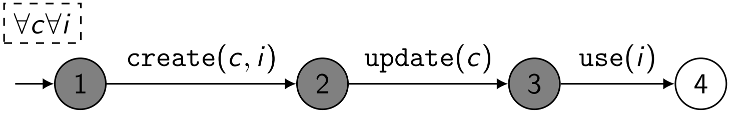
Illustrating Maximality



$\mathtt{update}(C).\mathtt{create}(C, A).\mathtt{use}(A).\mathtt{create}(B, A)$

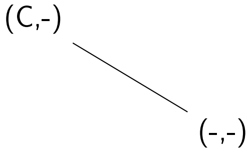
(-,-)

Illustrating Maximality

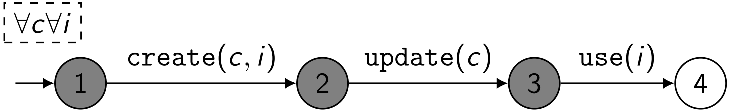


update(C).create(C, A).use(A).create(B, A)

(C,-) | update(C)

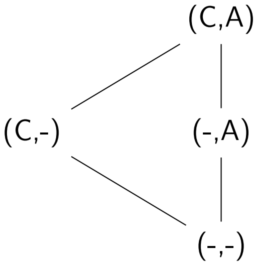


Illustrating Maximality

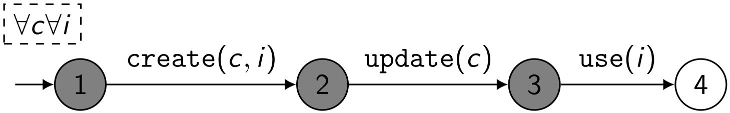


update(C).create(C, A).use(A).create(B, A)

(C,-)	update(C)
(-,A)	
(C,A)	update(C).create(C,A)

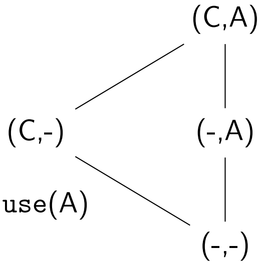


Illustrating Maximality



update(C).create(C, A).use(A).create(B, A)

(C,-)	update(C)
(-,A)	use(A)
(C,A)	update(C).create(C,A).use(A)



Illustrating Maximality

$$\boxed{\forall c \forall i} \quad \to (1) \xrightarrow{\texttt{create}(c,i)} (2) \xrightarrow{\texttt{update}(c)} (3) \xrightarrow{\texttt{use}(i)} (4)$$

$\texttt{update}(C).\texttt{create}(C,A).\texttt{use}(A).\texttt{create}(B,A)$

(C,-)	update(C)
(-,A)	use(A)
(B,-)	
(C,A)	update(C).create(C,A).use(A)
(B,A)	use(A).create(B,A)

$$\begin{tikzcd}
 & (C,A) \arrow[dl, no head] \arrow[d, no head] & (B,A) \arrow[dl, no head] \arrow[d, no head] \\
(C,-) \arrow[dr, no head] & (-,A) \arrow[d, no head] & (B,-) \arrow[dl, no head] \\
 & (-,-) &
\end{tikzcd}$$

RuleR

A rule has parameters and a list of rule terms of the form
event, *conditions* → *obligations*

$$
\begin{array}{rl}
\text{Start}\{ & \texttt{create}(c, i), !\text{Unsafe}(c, i) \rightarrow \text{Created}(c, i), \text{Start} \ \} \\
\text{Created}(c, i)\{ & \texttt{update}(c) \rightarrow \text{Unsafe}(c, i) \ \} \\
\text{Unsafe}(c, i)\{ & \texttt{use}(i) \rightarrow \text{Fail} \ \}
\end{array}
$$

. . . .

$\{\text{Start}\}$

RuleR

A rule has parameters and a list of rule terms of the form
event, *conditions* → *obligations*

$$\begin{array}{rl} \mathsf{Start}\{ & \mathtt{create}(c,i), !\mathsf{Unsafe}(c,i) \rightarrow \mathsf{Created}(c,i), \mathsf{Start}\ \} \\ \mathsf{Created}(c,i)\{ & \mathtt{update}(c) \rightarrow \mathsf{Unsafe}(c,i)\ \} \\ \mathsf{Unsafe}(c,i)\{ & \mathtt{use}(i) \rightarrow \mathsf{Fail}\ \} \end{array}$$

$\mathtt{create}(C,A).$ . . .

$$\{\mathsf{Start}\} \longrightarrow \{Start, Created(C,A)\}$$

RuleR

A rule has parameters and a list of rule terms of the form
event, *conditions* → *obligations*

$$\begin{array}{rl}
\text{Start}\{ & \texttt{create}(c,i), !\text{Unsafe}(c,i) \rightarrow \text{Created}(c,i), \text{Start}\ \} \\
\text{Created}(c,i)\{ & \texttt{update}(c) \rightarrow \text{Unsafe}(c,i)\ \} \\
\text{Unsafe}(c,i)\{ & \texttt{use}(i) \rightarrow \text{Fail}\ \}
\end{array}$$

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

$$\begin{array}{rl}
\{\text{Start}\} & \longrightarrow \{\mathit{Start}, \mathit{Created}(C,A)\} \\
 & \longrightarrow \{\mathit{Start}, \mathit{Created}(C,A), \mathit{Created}(C,B)\}
\end{array}$$

RuleR

A rule has parameters and a list of rule terms of the form
event, *conditions* $\rightarrow$ *obligations*

$$\begin{array}{rl} \text{Start}\{ & \texttt{create}(c,i), !\text{Unsafe}(c,i) \rightarrow \text{Created}(c,i), \text{Start}\ \} \\ \text{Created}(c,i)\{ & \texttt{update}(c) \rightarrow \text{Unsafe}(c,i)\ \} \\ \text{Unsafe}(c,i)\{ & \texttt{use}(i) \rightarrow \text{Fail}\ \} \end{array}$$

$\texttt{create}(C,A).\texttt{create}(C,B).\texttt{use}(A).$.

$$\begin{array}{rl} \{\text{Start}\} & \longrightarrow \{\mathit{Start}, \mathit{Created}(C,A)\} \\ & \longrightarrow \{\mathit{Start}, \mathit{Created}(C,A), \mathit{Created}(C,B)\} \\ & \longrightarrow \{\mathit{Start}, \mathit{Created}(C,A), \mathit{Created}(C,B)\} \end{array}$$

RuleR

A rule has parameters and a list of rule terms of the form
event, *conditions* $\rightarrow$ *obligations*

$$\begin{array}{rl}
\mathsf{Start}\{ & \mathtt{create}(c,i), !\mathsf{Unsafe}(c,i) \rightarrow \mathsf{Created}(c,i), \mathsf{Start}\ \} \\
\mathsf{Created}(c,i)\{ & \mathtt{update}(c) \rightarrow \mathsf{Unsafe}(c,i)\ \} \\
\mathsf{Unsafe}(c,i)\{ & \mathtt{use}(i) \rightarrow \mathsf{Fail}\ \}
\end{array}$$

$$\mathtt{create}(C,A).\mathtt{create}(C,B).\mathtt{use}(A).\mathtt{update}(C).$$

$$\begin{array}{rl}
\{\mathsf{Start}\} & \longrightarrow \{Start, Created(C,A)\} \\
& \longrightarrow \{Start, Created(C,A), Created(C,B)\} \\
& \longrightarrow \{Start, Created(C,A), Created(C,B)\} \\
& \longrightarrow \{Start, Unsafe(C,A), Unsafe(C,B)\}
\end{array}$$

RuleR

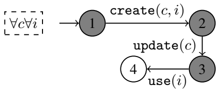
A rule has parameters and a list of rule terms of the form
event, *conditions* → *obligations*

$$\begin{array}{rl} \mathrm{Start}\{ & \texttt{create}(c,i), !\mathrm{Unsafe}(c,i) \rightarrow \mathrm{Created}(c,i), \mathrm{Start}\ \} \\ \mathrm{Created}(c,i)\{ & \texttt{update}(c) \rightarrow \mathrm{Unsafe}(c,i)\ \} \\ \mathrm{Unsafe}(c,i)\{ & \texttt{use}(i) \rightarrow \mathrm{Fail}\ \} \end{array}$$

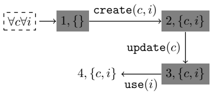
$$\texttt{create}(C,A).\texttt{create}(C,B).\texttt{use}(A).\texttt{update}(C).\texttt{use}(B)$$

$$\begin{array}{rl} \{\mathrm{Start}\} & \longrightarrow \{Start, Created(C,A)\} \\ & \longrightarrow \{Start, Created(C,A), Created(C,B)\} \\ & \longrightarrow \{Start, Created(C,A), Created(C,B)\} \\ & \longrightarrow \{Start, Unsafe(C,A), Unsafe(C,B)\} \\ & \longrightarrow \{Start, Unsafe(C,A), Fail\} = \{Fail\} \end{array}$$

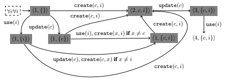
The Translation



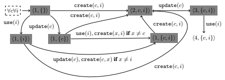
Input: QEA



Labelled Form



Domain-Explicit Form



Fresh-Variable Form

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

Output: Rule System

The idea is to get the QEA into a format where each state can be translated into a rule

Labelled States

Issue: Rules need to know which variables should be bound
Solution: Label states with the variables bound so far

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.

Each state is labelled with the union of variables that are seen on all paths to that state.

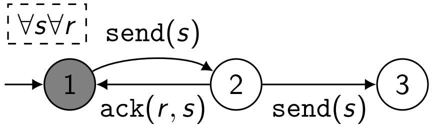
A QEA over labelled states is <u>well-labelled</u> if when $\langle q_2, S_2 \rangle$ is reachable from $\langle q_1, S_1 \rangle$ we have $S_1 \subseteq S_2$.

We can transform any QEA into a well-labelled QEA

Labelled States Example

Broadcast Property

For every sender s and receiver r, after s sends a message it should wait for an acknowledgement from r before sending again.



Labelled States Example

Broadcast Property

For every sender s and receiver r, after s sends a message it should wait for an acknowledgement from r before sending again.

$$\begin{tikzcd}
\boxed{\forall s \forall r} & & \\
\arrow[r] & 1 \arrow[r, bend left, "\mathtt{send}(s)"] & 2 \arrow[l, "\mathtt{ack}(r,s)"] \arrow[r, "\mathtt{send}(s)"'] & 3
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{\forall s \forall r} & & 3, \{s\} \arrow[r, "\mathtt{ack}(r,s)"'] & 3, \{r, s\} & 2, \{r, s\} \arrow[l, "\mathtt{send}(s)"'] \arrow[d, bend left, "\mathtt{ack}(r,s)"] \\
\arrow[r] & 1, \{\} \arrow[r, "\mathtt{send}(s)"] \arrow[rrrr, bend right, "\mathtt{ack}(r,s)"'] & 2, \{s\} \arrow[u, "\mathtt{send}(s)"] \arrow[rrr, "\mathtt{ack}(r,s)"] & & 1, \{r, s\} \arrow[u, "\mathtt{send}(s)"]
\end{tikzcd}$$

Domain-Explicit Form

Issue: Rules can only remember values (extend the quantification domain) when a transition occurs but in the QEA algorithm this can happen without any transition
Solution: Add transitions wherever the domain gets extended

These new transitions remain in the same state but extend the set of bound variables

We also have to add transitions for events where some of the variables are already bound <u>even if the value of that bound variable disagrees with the value in the observed event</u>.

Domain-Explicit Form Example

$$\begin{tikzcd}
\forall c \forall i \arrow[r] & 1, \{\} \arrow[r, "\texttt{create}(c, i)"] & 2, \{c, i\} \arrow[r, "\texttt{update}(c)"] & 3, \{c, i\} \arrow[r, "\texttt{use}(i)"] & 4, \{c, i\}
\end{tikzcd}$$

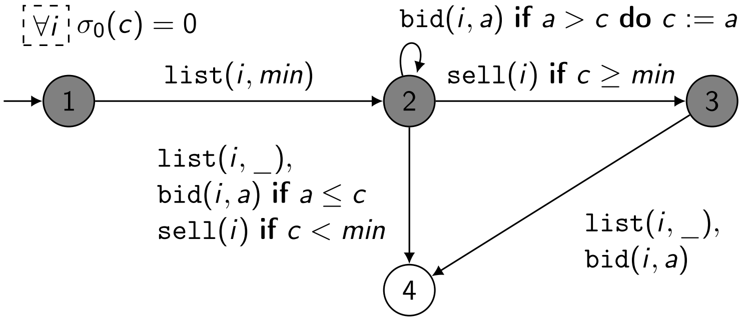
$$\begin{tikzcd}
\forall c \forall i \arrow[r] & \langle 1, \{\} \rangle \arrow[rr, "\texttt{create}(c, i)"] \arrow[dl, bend right, "\texttt{use}(i)"'] \arrow[d, bend left, "\texttt{update}(c)"] & & \langle 2, c, i\} \rangle \arrow[r, "\texttt{update}(c)"] & \langle 3, \{c, i\} \rangle \arrow[d, "\texttt{use}(i)"] \\
\langle 1, \{i\} \rangle \arrow[rrr, bend right, "\texttt{update}(c), \texttt{create}(c, x) \textbf{ if } x \neq i"] \arrow[urrr, bend right=60, "\texttt{create}(c, i)"'] & \langle 1, \{c\} \rangle \arrow[urr, bend left, "\texttt{create}(c, i)"] \arrow[rr, "\texttt{use}(i), \texttt{create}(x, i) \textbf{ if } x \neq c"] & & \langle 1, \{c, i\} \rangle \arrow[u, "\texttt{create}(c, i)"'] & \langle 4, \{c, i\} \rangle
\end{tikzcd}$$

This makes explicit what the monitoring algorithm does (without redundancy elimination) as it produces the full cross-product of all collections and iterators. We need to add redundancy elimination.

Local Variables: The Auction Bidding Example

Auction Bidding

After an item i is listed on an auction site with a reserve price min it cannot be relisted, all bids must be strictly increasing, and it can only be sold once this min price has been reached.


$\forall i$ $\sigma_0(c) = 0$
$\texttt{bid}(i,a)$ **if** $a > c$ **do** $c := a$
$\texttt{list}(i, min)$
$\texttt{sell}(i)$ **if** $c \geq min$
1
2
3
4
$\texttt{list}(i, \_)$,
$\texttt{bid}(i,a)$ **if** $a \leq c$
$\texttt{sell}(i)$ **if** $c < min$
$\texttt{list}(i, \_)$,
$\texttt{bid}(i,a)$

Fresh Variable Form

Issue: In a rule, any variables appearing in the parameter list of a rule get instantiated in the transition
Solution: Rename local variables in transitions

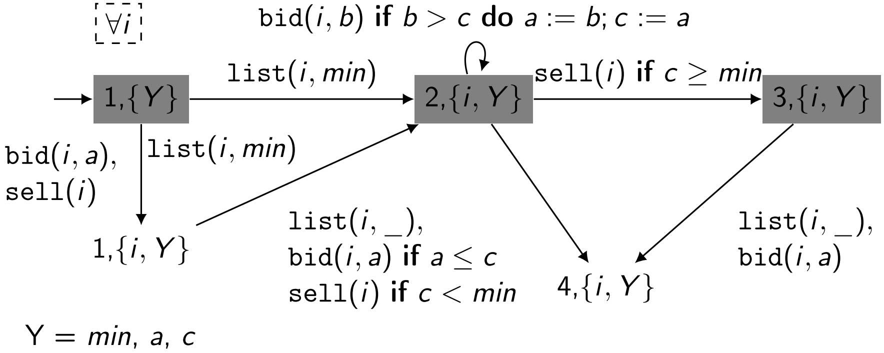
Without the renaming we would end up producing the rule

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

which would not work

To avoid an explosion of variables we can keep the variable local to the transition, such variables do not need to be added to the set of bound variables at states as they do not need to be remembered.

Fresh Variable Form



To Rules

All states are translated into rules and all transitions are translated into rule terms

For a transition there are two cases

1. <u>The transition does not bound new variables.</u> The translation is a straightforward translation of event, guard, and action.

2. <u>The transition bounds new variables.</u> Then we need to check if the current rule instance is maximal. We do this by adding negations for all rules that could bound more variables.

The Result: UnsafeIter Property

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

The Result: UnsafeIter Property

$$
\begin{array}{ll}
r_1 & \left\{ \begin{array}{l}
\mathtt{update}(c), !r_1(c), !r_1(i), !r_1(c,i), !r_2(c,i), !r_3(c,i), !r_4(c,i) \rightarrow r_1, r_1(c) \\
\mathtt{create}(c,i), !r_1(c), !r_1(i), !r_1(c,i), !r_2(c,i), !r_3(c,i), !r_4(c,i) \rightarrow r_1, r_2(c,i) \\
\mathtt{use}(i), !r_1(c), !r_1(i), !r_1(c,i), !r_2(c,i), !r_3(c,i), !r_4(c,i) \rightarrow r_1, r_1(i)
\end{array} \right\} \\
r_1(c) & \left\{ \begin{array}{l}
\mathtt{create}(c,i), !r_1(c,i), !r_2(c,i), !r_3(c,i), !r_4(c,i) \rightarrow r_1(c), r_2(c,i) \\
\mathtt{use}(i), !r_1(c,i), !r_2(c,i), !r_3(c,i), !r_4(c,i) \rightarrow r_1(c), r_1(c,i) \\
\mathtt{create}(c_p,i), c \neq c_p, !r_1(c,i), !r_2(c,i), !r_3(c,i), !r_4(c,i) \rightarrow r_1(c), r_1(c,i)
\end{array} \right\} \\
r_1(i) & \left\{ \begin{array}{l}
\mathtt{create}(c,i_p), i \neq i_p, !r_1(c,i), !r_2(c,i), !r_3(c,i), !r_4(c,i) \rightarrow r_1(i), r_1(c,i) \\
\mathtt{update}(c), !r_1(c,i), !r_2(c,i), !r_3(c,i), !r_4(c,i) \rightarrow r_1(i), r_1(c,i) \\
\mathtt{create}(c,i), !r_1(c,i), !r_2(c,i), !r_3(c,i), !r_4(c,i) \rightarrow r_1(i), r_2(c,i)
\end{array} \right\} \\
r_1(c,i) & \left\{ \mathtt{create}(c,i) \rightarrow r_2(c,i) \right\} \\
r_2(c,i) & \left\{ \mathtt{update}(c) \rightarrow r_3(c,i) \right\} \\
r_3(c,i) & \left\{ \mathtt{use}(i) \rightarrow r_4(c,i) \right\} \\
r_4(c,i) & \{\}
\end{array}
$$

The Result: Broadcast Property

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

The Result: Broadcast Property

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

The Result: Broadcast Property

For every sender s and receiver r, after s sends a message it should wait for an acknowledgement from r before sending again.

$$\begin{array}{rl}
\mathsf{r_1} & \left\{ \begin{array}{l} \mathtt{ack}(r,s), !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_2}(s), !\mathsf{r_3}(r,s), !\mathsf{r_3}(s) \to \mathsf{r_1}, \mathsf{r_1}(r,s) \\ \mathtt{send}(s), !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_2}(s), !\mathsf{r_3}(r,s), !\mathsf{r_3}(s) \to \mathsf{r_1}, \mathsf{r_2}(s) \end{array} \right\} \\
\mathsf{r_1}(r,s) & \left\{ \mathtt{send}(s) \to \mathsf{r_2}(r,s) \right\} \\
\mathsf{r_2}(s) & \left\{ \begin{array}{l} \mathtt{send}(s) \to \mathsf{r_3}(s) \\ \mathtt{ack}(r,s_p), s \neq s_p, !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_3}(r,s) \to \mathsf{r_2}(s), \mathsf{r_2}(r,s) \\ \mathtt{ack}(r,s), !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_3}(r,s) \to \mathsf{r_2}(s), \mathsf{r_1}(r,s) \end{array} \right\} \\
\mathsf{r_2}(r,s) & \left\{ \begin{array}{l} \mathtt{send}(s) \to \mathsf{r_3}(r,s) \\ \mathtt{ack}(r,s) \to \mathsf{r_1}(r,s) \end{array} \right\} \\
\mathsf{r_3}(s) & \left\{ \begin{array}{l} \mathtt{ack}(r,s_p), s \neq s_p, !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_3}(r,s) \to \mathsf{r_3}(s), \mathsf{r_3}(r,s) \\ \mathtt{ack}(r,s), !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_3}(r,s) \to \mathsf{r_3}(s), \mathsf{r_3}(r,s) \end{array} \right\} \\
\mathsf{r_3}(r,s) & \{\}
\end{array}$$

Hand-written version from before we started the process:

$$\begin{array}{rl}
\mathsf{Start} & \left\{ \begin{array}{l} \mathtt{send}(s), !\mathsf{S}(s) \to \mathsf{S}(s), \mathsf{Start} \\ \mathtt{send}(s), !\mathsf{S}(s), R(r) \to \mathsf{Unsafe}(r,s), \mathsf{Start} \\ \mathtt{send}(s), \mathsf{S}(s) \to \mathsf{Fixed} \\ \mathtt{ack}(r,s), !\mathsf{R}(r) \to \mathsf{R}(r), \mathsf{Start} \end{array} \right\} \\
\mathsf{Fixed} & \left\{ \begin{array}{l} \mathtt{ack}(r,s), !\mathsf{R}(r) \to \mathsf{Fail} \\ \mathtt{send}(s), !\mathsf{S}(s), \mathsf{R}(r) \to \mathsf{S}(s), \mathsf{Unsafe}(r,s), \mathsf{Fixed} \end{array} \right\} \\
\mathsf{S}(s) & \left\{ \begin{array}{l} \mathtt{send}(s), \mathsf{R}(r) \to \mathsf{Unsafe}(r,s), \mathsf{S}(s) \\ \mathtt{ack}(r,s'), !\mathsf{R}(r), s \neq s' \to \mathsf{Unsafe}(r,s), \mathsf{S}(s) \end{array} \right\} \\
\mathsf{Unsafe}(r,s) & \left\{ \begin{array}{l} \mathtt{send}(s) \to \mathsf{Fail} \\ \mathtt{ack}(r,s) \to \textit{empty} \end{array} \right\} \\
\mathsf{R}(r) & \{\}
\end{array}$$

The Result: Broadcast Property

For every sender s and receiver r, after s sends a message it should wait for an acknowledgement from r before sending again.

$$\begin{array}{rl}
\mathsf{r_1} & \left\{ \begin{array}{l} \mathtt{ack}(r,s), !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_2}(s), !\mathsf{r_3}(r,s), !\mathsf{r_3}(s) \to \mathsf{r_1}, \mathsf{r_1}(r,s) \\ \mathtt{send}(s), !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_2}(s), !\mathsf{r_3}(r,s), !\mathsf{r_3}(s) \to \mathsf{r_1}, \mathsf{r_2}(s) \end{array} \right\} \\
\mathsf{r_1}(r,s) & \left\{ \mathtt{send}(s) \to \mathsf{r_2}(r,s) \right\} \\
\mathsf{r_2}(s) & \left\{ \begin{array}{l} \mathtt{send}(s) \to \mathsf{r_3}(s) \\ \mathtt{ack}(r,s_p), s \neq s_p, !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_3}(r,s) \to \mathsf{r_2}(s), \mathsf{r_2}(r,s) \\ \mathtt{ack}(r,s), !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_3}(r,s) \to \mathsf{r_2}(s), \mathsf{r_1}(r,s) \end{array} \right\} \\
\mathsf{r_2}(r,s) & \left\{ \begin{array}{l} \mathtt{send}(s) \to \mathsf{r_3}(r,s) \\ \mathtt{ack}(r,s) \to \mathsf{r_1}(r,s) \end{array} \right\} \\
\mathsf{r_3}(s) & \left\{ \begin{array}{l} \mathtt{ack}(r,s_p), s \neq s_p, !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_3}(r,s) \to \mathsf{r_3}(s), \mathsf{r_3}(r,s) \\ \mathtt{ack}(r,s), !\mathsf{r_1}(r,s), !\mathsf{r_2}(r,s), !\mathsf{r_3}(r,s) \to \mathsf{r_3}(s), \mathsf{r_3}(r,s) \end{array} \right\} \\
\mathsf{r_3}(r,s) & \{\}
\end{array}$$

Hand-written version from before we started the process:

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

The Result: Broadcast Property

For every sender s and receiver r, after s sends a message it should wait for an acknowledgement from r before sending again.

$$
\begin{array}{rl}
\mathbf{r_1} & \left\{\begin{array}{l} \mathtt{ack}(r,s), !\mathbf{r_1}(r,s), !\mathbf{r_2}(r,s), !\mathbf{r_2}(s), !\mathbf{r_3}(r,s), !\mathbf{r_3}(s) \to \mathbf{r_1}, \mathbf{r_1}(r,s) \\ \mathtt{send}(s), !\mathbf{r_1}(r,s), !\mathbf{r_2}(r,s), !\mathbf{r_2}(s), !\mathbf{r_3}(r,s), !\mathbf{r_3}(s) \to \mathbf{r_1}, \mathbf{r_2}(s) \end{array}\right\} \\
\mathbf{r_1}(r,s) & \left\{ \mathtt{send}(s) \to \mathbf{r_2}(r,s) \right\} \\
\mathbf{r_2}(s) & \left\{\begin{array}{l} \mathtt{send}(s) \to \mathbf{r_3}(s) \\ \mathtt{ack}(r,s_p), s \neq s_p, !\mathbf{r_1}(r,s), !\mathbf{r_2}(r,s), !\mathbf{r_3}(r,s) \to \mathbf{r_2}(s), \mathbf{r_2}(r,s) \\ \mathtt{ack}(r,s), !\mathbf{r_1}(r,s), !\mathbf{r_2}(r,s), !\mathbf{r_3}(r,s) \to \mathbf{r_2}(s), \mathbf{r_1}(r,s) \end{array}\right\} \\
\mathbf{r_2}(r,s) & \left\{\begin{array}{l} \mathtt{send}(s) \to \mathbf{r_3}(r,s) \\ \mathtt{ack}(r,s) \to \mathbf{r_1}(r,s) \end{array}\right\} \\
\mathbf{r_3}(s) & \left\{\begin{array}{l} \mathtt{ack}(r,s_p), s \neq s_p, !\mathbf{r_1}(r,s), !\mathbf{r_2}(r,s), !\mathbf{r_3}(r,s) \to \mathbf{r_3}(s), \mathbf{r_3}(r,s) \\ \mathtt{ack}(r,s), !\mathbf{r_1}(r,s), !\mathbf{r_2}(r,s), !\mathbf{r_3}(r,s) \to \mathbf{r_3}(s), \mathbf{r_3}(r,s) \end{array}\right\} \\
\mathbf{r_3}(r,s) & \{\}
\end{array}
$$

Hand-written version from before we started the process:

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

The Result: Broadcast Property

For every sender s and receiver r, after s sends a message it should wait for an acknowledgement from r before sending again.

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

Hand-written version from before we started the process:

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

Properties

Theorem

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

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

then for any valuation θ

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

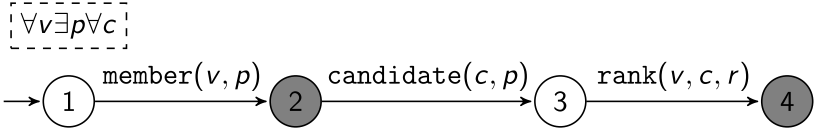
The translation is decidable; any universal QEA can be translated to a rule system (which is neither unique nor minimal; no good notion of minimality exists).

The size of the resulting rule system is potentially $O(|Q| \times 2^{|X|})$ due to the well-labelled translation introducing new states.

Existential Quantification

Existential quantification is difficult for rule-based systems as the universal quantification is implicit in the requirement for stored information to be consistent

For the following property, the best we could do for a rule system was to record all information, add a special last event, and then check the property on stored information



One possible solution would be to add a new kind of non-determinism

Future Work

Check efficiency of translated rule systems in RuleR

Embed redundancy elimination from monitoring algorithm in translation

Original RuleR work embedded LTL in RuleR, extend to first-order

From rule-systems to first-order temporal logic

From a star-free fragment of QEA to LTL (using nested $\exists$ for local variables)

From MFTL to QEA

Relationship between various first-order temporal logics?

Stream-based languages?

Conclusion

In this talk I have:

- Motivated and described our research vision
- Described a translation from QEA to rule systems
- Given some examples and properties
- Didn't mention that the translation is implemented in Scala

Thank you for listening

Any questions?