From Parametric Trace Slicing to Rule Systems

Giles Reger       David Rydeheard

University of Manchester, Manchester, UK

November 11, 2018
Spot the Difference

\[ \square \forall i. \left( \text{next}(i) \rightarrow \exists m, c. \left( \neg \text{update}(m) \land \text{iterator}(c, i) \land \text{create}(m, c) \right) \right) \]

\[
\begin{align*}
\text{createdC} & = \text{create} \cup \text{createdC}[-1|\emptyset] \\
\text{createdI} & = (\text{iterator} \bowtie \text{createdC}) \cup \text{createdI}[-1|\emptyset] \\
\text{updated} & = (\text{update} \bowtie \text{createdI}) \cup \text{updated}[-1|\emptyset] \\
\text{ok} & = \text{next} /\subseteq \pi_{<i}(\text{updated})
\end{align*}
\]

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

r1: create(m, c) \rightarrow \text{createdC}(m, c)

r2: create(m, c), iterator(c, i) \rightarrow \text{createdI}(m, i)

r3: create(m, c), update(m) \rightarrow \text{updated}(i)

r4: updated(i), next(i) \rightarrow \text{error}
}
Some Terminology: Parametric RV

Runtime Verification Problem
Given a trace $\tau$ and a specification $\varphi$ 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 $\varphi$ (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\(^{FO}\), 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

- **Rule-based**
  - Core RuleR

- **Temporal Logic-based**
  - $\mathcal{F}(\text{future FO LTL})$

- **Slicing-based**
  - $\forall$ QEA (locals)
  - $\forall \exists$ det QEA (no locals)

- **Stream-based**
  - ?
The Landscape

- Rule-based: Core RuleR
- Temporal Logic-based: \( \mathcal{F} \) (future FO LTL)
- Slicing-based: \( \forall QEA \) (locals), \( \exists \) det QEA (no locals)
- Stream-based: ?
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

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)
Motivation Two Languages The Translation Properties Discussion

Quantified Event Automata

\[ \forall c \forall i \]

\[ \text{create}(c, i) \quad \text{update}(c) \quad \text{use}(i) \]

\[ \text{create}(C, A).\text{create}(C, B).\text{use}(A).\text{update}(C).\text{use}(B) \]

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

\[ \forall c \forall i \]

\begin{align*}
&\text{create}(c, i) \quad \text{update}(c) \quad \text{use}(i) \\
&\text{create}(C, A), \text{create}(C, B), \text{use}(A), \text{update}(C), \text{use}(B)
\end{align*}

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 $\theta_j \sqsubseteq \theta_k$ 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 $(a * M) = N_m$ where $N_m$ is iteratively defined as follows for $i \in [1, m]$ and $N_0 = \bot$

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

with additions defined in terms of extensions not already present:

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

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

\[ \forall c \forall i \]

1. create\((c, i)\)  
2. update\((c)\)  
3. use\((i)\)  
4. 

update\((C)\).create\((C, A)\).use\((A)\).create\((B, A)\)
Illustrating Maximality

∀c∀i create(c, i) update(c) use(i)

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

(C,-) update(C)

(C,-) (C,-) (-,-)
Illustrating Maximality

∀c∀i

1. create(c, i) → 2. update(c) → 3. use(i) → 4

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

(C,A)

(C,-)  update(C)

(-,A)   (C,-)  (-,A)

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

(-,-)
Illustrating Maximality

∀c∀i

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

(C,A)

(C,-) update(C) 

(-,A) use(A) 

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

(-,-)
Motivation Two Languages The Translation Properties Discussion

Illustrating Maximality

∀c∀i

create(c, i) → update(c) → use(i) → 4

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

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

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

\[
\text{Start}\{\ \text{create}(c, i), \! \text{Unsafe}(c, i) \rightarrow \text{Created}(c, i), \text{Start} \} \\
\text{Created}(c, i)\{\ \text{update}(c) \rightarrow \text{Unsafe}(c, i) \} \\
\text{Unsafe}(c, i)\{\ \text{use}(i) \rightarrow \text{Fail} \}
\]

\[
\{\text{Start}\}
\]
RuleR

A rule has parameters and a list of rule terms of the form 
\textit{event}, \textit{conditions} \rightarrow \textit{obligations}

\begin{align*}
\text{Start}\{ & \text{create}(c, i), !\text{Unsafe}(c, i) \rightarrow \text{Created}(c, i), \text{Start} \} \\
\text{Created}(c, i)\{ & \text{update}(c) \rightarrow \text{Unsafe}(c, i) \} \\
\text{Unsafe}(c, i)\{ & \text{use}(i) \rightarrow \text{Fail} \}
\end{align*}

\text{create}(C, A).

\{\text{Start}\} \rightarrow \{\text{Start}, \text{Created}(C, A)\}
RuleR

A rule has parameters and a list of rule terms of the form
\textit{event, conditions} \rightarrow \textit{obligations}

\begin{align*}
\text{Start}\{ & \text{create}(c, i), !\text{Unsafe}(c, i) \rightarrow \text{Created}(c, i), \text{Start} \} \\
\text{Created}(c, i)\{ & \text{update}(c) \rightarrow \text{Unsafe}(c, i) \} \\
\text{Unsafe}(c, i)\{ & \text{use}(i) \rightarrow \text{Fail} \}
\end{align*}

\begin{align*}
\text{create}(C, A) . \text{create}(C, B) .
\end{align*}

\begin{align*}
\{ \text{Start} \} & \rightarrow \{ \text{Start}, \text{Created}(C, A) \} \\
& \rightarrow \{ \text{Start}, \text{Created}(C, A), \text{Created}(C, B) \}
\end{align*}
RuleR

A rule has parameters and a list of rule terms of the form
\textit{event, conditions} \rightarrow \textit{obligations}

\begin{align*}
\text{Start} & \{ \text{create}(c, i), \lnot \text{Unsafe}(c, i) \rightarrow \text{Created}(c, i), \text{Start} \} \\
\text{Created}(c, i) & \{ \text{update}(c) \rightarrow \text{Unsafe}(c, i) \} \\
\text{Unsafe}(c, i) & \{ \text{use}(i) \rightarrow \text{Fail} \}
\end{align*}

\text{create}(C, A). \text{create}(C, B). \text{use}(A). \quad .

\{ \text{Start} \} \quad \rightarrow \quad \{ \text{Start}, \text{Created}(C, A) \} \\
\quad \rightarrow \quad \{ \text{Start}, \text{Created}(C, A), \text{Created}(C, B) \} \\
\quad \rightarrow \quad \{ \text{Start}, \text{Created}(C, A), \text{Created}(C, B) \}
A rule has parameters and a list of rule terms of the form \textit{event}, \textit{conditions} $\rightarrow$ \textit{obligations}

RuleR


create(c, i), !Unsafe(c, i) \rightarrow \textit{Created}(c, i), \textit{Start}

Created(c, i){ \quad \textit{update}(c) \rightarrow \textit{Unsafe}(c, i) }

Unsafe(c, i){ \quad \textit{use}(i) \rightarrow \textit{Fail} }


\{\textit{Start}\} \quad \rightarrow\quad \{\textit{Start}, \textit{Created}(C, A)\}
\rightarrow\quad \{\textit{Start}, \textit{Created}(C, A), \textit{Created}(C, B)\}
\rightarrow\quad \{\textit{Start}, \textit{Created}(C, A), \textit{Created}(C, B)\}
\rightarrow\quad \{\textit{Start}, \textit{Unsafe}(C, A), \textit{Unsafe}(C, B)\}
RuleR

A rule has parameters and a list of rule terms of the form
\textit{event, conditions} \rightarrow \textit{obligations}

\begin{align*}
\text{Start} & \{ \text{create}(c, i), !\text{Unsafe}(c, i) \rightarrow \text{Created}(c, i), \text{Start} \} \\
\text{Created}(c, i) & \{ \text{update}(c) \rightarrow \text{Unsafe}(c, i) \} \\
\text{Unsafe}(c, i) & \{ \text{use}(i) \rightarrow \text{Fail} \}
\end{align*}

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

\begin{align*}
\{\text{Start}\} & \rightarrow \{\text{Start}, \text{Created}(C, A)\} \\
& \rightarrow \{\text{Start}, \text{Created}(C, A), \text{Created}(C, B)\} \\
& \rightarrow \{\text{Start}, \text{Created}(C, A), \text{Created}(C, B)\} \\
& \rightarrow \{\text{Start}, \text{Unsafe}(C, A), \text{Unsafe}(C, B)\} \\
& \rightarrow \{\text{Start}, \text{Unsafe}(C, A), \text{Fail}\} = \{\text{Fail}\}
\end{align*}
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 well-labelled if when \( \langle q_2, S_2 \rangle \) is reachable from \( \langle q_1, S_1 \rangle \) we have \( S_1 \subseteq S_2 \).

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.

$\forall s \forall r : \text{send}(s)$

1

$\text{ack}(r, s)$

2

$\text{send}(s)$

3

∀$s$∀$r$ : \text{send}(s)$

∀$s$∀$r$ : \text{send}(s)$
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.
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 even if the value of that bound variable disagrees with the value in the observed event.
Motivation Two Languages The Translation Properties Discussion

Domain-Explicit Form Example

∀c∀i
create(c, i) update(c) use(i)
1, {} 2, {c, i} 3, {c, i} 4, {c, i}

∀c∀i
<1, {}> create(c, i) update(c) use(i)
<1, {i}> <2, {c, i}> <3, {c, i}>
<1, {c}> create(c, i) create(c, i) create(c, i)
update(c) create(x, i) if x ≠ c
<1, {c, i}> <1, {c, i}> <3, {c, i}>

update(c), create(c, x) if x ≠ i
create(c, i)

∀c∀i
<1, {}> create(c, i) update(c) use(i)
<1, {i}> <2, {c, i}> <3, {c, i}>
<1, {c}> create(c, i) create(c, i) create(c, i)
update(c) create(x, i) if x ≠ c
<1, {c, i}> <1, {c, i}> <3, {c, i}>

update(c), create(c, x) if x ≠ i
create(c, i)

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.
Motivation Two Languages The Translation Properties Discussion

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.

\[ \forall i \sigma_0(c) = 0 \]

\[
\begin{align*}
\text{list}(i, min) & \quad \text{bid}(i, a) \text{ if } a > c \text{ do } c := a \\
\text{list}(i, _) , & \quad \text{sell}(i) \text{ if } c \geq min \\
\text{bid}(i, a) \text{ if } a \leq c & \\
\text{sell}(i) \text{ if } c < min & \\
\text{list}(i, _) , & \\
\text{bid}(i, a) &
\end{align*}
\]
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

$$r_2(i, \text{min}, c, a)\{\text{bid}(i, a), a > c \rightarrow r_2(i, \text{min}, a, a)\}$$

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.
Motivation Two Languages The Translation Properties Discussion

Fresh Variable Form

∀ i list(i, min) bid(i, b) if b > c do a := b; c := a

list(i, min) sell(i) if c ≥ min

1, {i, Y}  2, {i, Y}  3, {i, Y}

bid(i, a), list(i, min) sell(i)

1, {i, Y}

bid(i, a), list(i, _), bid(i, a) if a ≤ c
sell(i) if c < min

4, {i, Y}

list(i, _), bid(i, a)

Y = min, a, c
To Rules

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

For a transition there are two cases:

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

2. The transition bounds new variables. 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

\[
\begin{align*}
\text{update}(c), & \text{!}r_1(c), \text{!}r_1(i), \text{!}r_2(c, i), \text{!}r_3(c, i), \text{!}r_4(c, i) \rightarrow r_1, r_1(c) \\
\text{create}(c, i), & \text{!}r_1(c), \text{!}r_1(i), \text{!}r_2(c, i), \text{!}r_3(c, i), \text{!}r_4(c, i) \rightarrow r_1, r_2(c, i) \\
\text{use}(i), & \text{!}r_1(c), \text{!}r_1(i), \text{!}r_2(c, i), \text{!}r_3(c, i), \text{!}r_4(c, i) \rightarrow r_1, r_1(i) \\
\text{create}(c, i), & \text{!}r_1(c, i), \text{!}r_2(c, i), \text{!}r_3(c, i), \text{!}r_4(c, i) \rightarrow r_1(c), r_2(c, i) \\
\text{use}(i), & \text{!}r_1(c, i), \text{!}r_2(c, i), \text{!}r_3(c, i), \text{!}r_4(c, i) \rightarrow r_1(c), r_1(c, i) \\
\text{create}(c, p), & c \neq c_p, \text{!}r_1(c, i), \text{!}r_2(c, i), \text{!}r_3(c, i), \text{!}r_4(c, i) \rightarrow r_1(c), r_1(c, i) \\
\text{create}(c, i_p), & i \neq i_p, \text{!}r_1(c, i), \text{!}r_2(c, i), \text{!}r_3(c, i), \text{!}r_4(c, i) \rightarrow r_1(i), r_1(c, i) \\
\text{update}(c), & \text{!}r_1(c, i), \text{!}r_2(c, i), \text{!}r_3(c, i), \text{!}r_4(c, i) \rightarrow r_1(i), r_1(c, i) \\
\text{create}(c, i), & \text{!}r_1(c, i), \text{!}r_2(c, i), \text{!}r_3(c, i), \text{!}r_4(c, i) \rightarrow r_1(i), r_2(c, i) \\
\text{create}(c, i) & \rightarrow r_2(c, i) \\
\text{update}(c) & \rightarrow r_3(c, i) \\
\text{use}(i) & \rightarrow r_4(c, i) \\
\end{align*}
\]
The Result: UnsafeIter Property

\[
\begin{align*}
\text{update}(c), \, \text{!}r_1(c), \, \text{!}r_1(i), \, \text{!}r_1(c, i), \, \text{!}r_2(c, i), \, \text{!}r_3(c, i), \, \text{!}r_4(c, i) & \rightarrow r_1, r_1(c) \\
\text{create}(c, i), \, \text{!}r_1(c), \, \text{!}r_1(i), \, \text{!}r_1(c, i), \, \text{!}r_2(c, i), \, \text{!}r_3(c, i), \, \text{!}r_4(c, i) & \rightarrow r_1, r_2(c, i) \\
\text{use}(i), \, \text{!}r_1(c), \, \text{!}r_1(i), \, \text{!}r_1(c, i), \, \text{!}r_2(c, i), \, \text{!}r_3(c, i), \, \text{!}r_4(c, i) & \rightarrow r_1, r_1(i) \\
\text{create}(c, i), \, \text{!}r_1(c, i), \, \text{!}r_2(c, i), \, \text{!}r_3(c, i), \, \text{!}r_4(c, i) & \rightarrow r_1(c), r_2(c, i) \\
\text{use}(i), \, \text{!}r_1(c, i), \, \text{!}r_2(c, i), \, \text{!}r_3(c, i), \, \text{!}r_4(c, i) & \rightarrow r_1(c), r_1(c, i) \\
\text{create}(c_p, i), \, c \neq c_p, \, \text{!}r_1(c, i), \, \text{!}r_2(c, i), \, \text{!}r_3(c, i), \, \text{!}r_4(c, i) & \rightarrow r_1(c), r_1(c, i) \\
\text{create}(c, i_p), \, i \neq i_p, \, \text{!}r_1(c, i), \, \text{!}r_2(c, i), \, \text{!}r_3(c, i), \, \text{!}r_4(c, i) & \rightarrow r_1(i), r_1(c, i) \\
\text{update}(c), \, \text{!}r_1(c, i), \, \text{!}r_2(c, i), \, \text{!}r_3(c, i), \, \text{!}r_4(c, i) & \rightarrow r_1(i), r_1(c, i) \\
\text{create}(c, i), \, \text{!}r_1(c, i), \, \text{!}r_2(c, i), \, \text{!}r_3(c, i), \, \text{!}r_4(c, i) & \rightarrow r_1(i), r_2(c, i) \\
\text{create}(c, i) & \rightarrow r_2(c, i) \\
\text{update}(c) & \rightarrow r_3(c, i) \\
\text{use}(i) & \rightarrow r_4(c, i)
\end{align*}
\]
The Result: Broadcast Property

\[\begin{align*}
&\text{r}_1 \{ \text{ack}(r, s), \text{send}(r, s), \text{ack}(s, r), \text{send}(s, r) \rightarrow \text{r}_1, \text{r}_1(r, s) \} \\
&\text{r}_1(r, s) \{ \text{send}(s) \rightarrow \text{r}_2(r, s) \} \\
&\text{r}_2(s) \{ \text{ack}(r, s), \text{send}(s) \rightarrow \text{r}_3(r, s) \} \\
&\text{r}_2(r, s) \{ \text{ack}(r, s) \rightarrow \text{r}_1(r, s) \} \\
&\text{r}_3(s) \{ \text{ack}(r, s), \text{send}(s) \rightarrow \text{r}_3(r, s) \} \\
&\text{r}_3(r, s) \{ \} \\
\end{align*}\]
The Result: Broadcast Property

\[
\begin{align*}
  r_1 & \quad \{ \text{ack}(r, s), !r_1(r, s), !r_2(r, s), !r_2(s), !r_3(r, s), !r_3(s) \rightarrow r_1, r_1(r, s) \} \\
  r_1(r, s) & \quad \{ \text{send}(s), !r_1(r, s), !r_2(r, s), !r_2(s), !r_3(r, s), !r_3(s) \rightarrow r_1, r_2(s) \} \\
  r_2(s) & \quad \{ \text{send}(s) \rightarrow r_2(r, s) \} \\
  r_2(r, s) & \quad \{ \text{send}(s) \rightarrow r_3(s) \} \\
  r_3(s) & \quad \{ \text{ack}(r, s), s \neq s_p, !r_1(r, s), !r_2(r, s), !r_2(s), !r_3(r, s) \rightarrow r_3(s), r_2(s), r_2(r, s) \} \\
  r_3(r, s) & \quad \{ \text{ack}(r, s), !r_1(r, s), !r_2(r, s), !r_3(r, s) \rightarrow r_3(s), r_3(r, s) \} \\
\end{align*}
\]
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.

$$
\begin{align*}
\text{send}(s) \rightarrow r_2(r, s) \\
\text{send}(s) \rightarrow r_3(s) \\
\text{send}(s) \rightarrow r_1(r, s) \\
\text{ack}(r, s) \rightarrow S(s) \rightarrow S(s), \text{Start} \\
\text{ack}(r, s) \rightarrow R(r) \rightarrow \text{Unsafe}(r, s), \text{Start} \\
\text{ack}(r, s) \rightarrow \text{Fixed} \\
\text{ack}(r, s) \rightarrow \text{Fail} \\
\text{send}(s), R(r) \rightarrow S(s), \text{Unsafe}(r, s), \text{Fixed} \\
\text{send}(s), R(r) \rightarrow \text{Unsafe}(r, s), S(s) \\
\text{ack}(r, s) \rightarrow \text{Unsafe}(r, s), S(s) \\
\text{send}(s) \rightarrow \text{Fail} \\
\text{ack}(r, s) \rightarrow \text{empty}
\end{align*}
$$
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.

\[
\begin{align*}
    &r_1 \quad \{ \begin{array}{l}
    \text{send}(s), \neg r_1(s), !r_2(s), !r_2(s), !r_3(s) \rightarrow r_1, r_1(s) \\
    \text{send}(s), \neg r_1(s), !r_2(s), !r_2(s), !r_3(s) \rightarrow r_1, r_2(s)
    \end{array} \} \\
    &r_1(r, s) \quad \{ \begin{array}{l}
    \text{send}(s) \rightarrow r_2(s)
    \end{array} \} \\
    &r_2(s) \quad \{ \begin{array}{l}
    \text{send}(s) \rightarrow r_3(s) \\
    \text{ack}(r, s), \neg s_p, \neg r_1(r, s), !r_2(r, s), !r_3(r, s) \rightarrow r_2(s), r_2(s) \\
    \text{ack}(r, s), \neg r_1(r, s), !r_2(r, s), !r_3(r, s) \rightarrow r_2(s), r_1(r, s)
    \end{array} \} \\
    &r_2(r, s) \quad \{ \begin{array}{l}
    \text{send}(s) \rightarrow r_3(s) \\
    \text{ack}(r, s) \rightarrow r_1(r, s)
    \end{array} \} \\
    &r_3(s) \quad \{ \begin{array}{l}
    \text{ack}(r, s), \neg s_p, \neg r_1(r, s), !r_2(r, s), !r_3(r, s) \rightarrow r_3(s), r_3(s) \\
    \text{ack}(r, s), \neg r_1(r, s), !r_2(r, s), !r_3(r, s) \rightarrow r_3(s), r_3(r, s)
    \end{array} \} \\
    &r_3(r, s) \quad \{ \}
\end{align*}
\]

Hand-written version from before we started the process:

\[
\begin{align*}
    &\text{Start} \quad \{ \begin{array}{l}
    \text{send}(s), !S(s) \rightarrow S(s), \text{Start} \\
    \text{send}(s), !S(s), R(r) \rightarrow \text{Unsafe}(r, s), \text{Start} \\
    \text{send}(s), S(s) \rightarrow \text{Fixed} \\
    \text{ack}(r, s), \neg R(r) \rightarrow R(r), \text{Start}
    \end{array} \} \\
    &\text{Fixed} \quad \{ \begin{array}{l}
    \text{ack}(r, s), \neg R(r) \rightarrow \text{Fail} \\
    \text{send}(s), !S(s), R(r) \rightarrow S(s), \text{Unsafe}(r, s), \text{Fixed} \\
    \text{send}(s), R(r) \rightarrow \text{Unsafe}(r, s), S(s)
    \end{array} \} \\
    &S(s) \quad \{ \begin{array}{l}
    \text{ack}(r, s'), \neg R(r), \neg s' \rightarrow \text{Unsafe}(r, s), S(s)
    \end{array} \} \\
    &\text{Unsafe}(r, s) \quad \{ \begin{array}{l}
    \text{send}(s) \rightarrow \text{Fail}
    \end{array} \} \\
    &R(r) \quad \{ \begin{array}{l}
    \text{ack}(r, s) \rightarrow \text{empty}
    \end{array} \}
\end{align*}
\]
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.

\[
\begin{align*}
\text{Start} & \quad \{ \text{send}(s), !S(s) \rightarrow S(s), \text{Start} \\
& \quad \text{send}(s), !S(s), R(r) \rightarrow \text{Unsafe}(r, s), \text{Start} \\
& \quad \text{send}(s), S(s) \rightarrow \text{Fixed} \\
& \quad \text{ack}(r, s), !R(r) \rightarrow R(r), \text{Start} \\
& \quad \text{ack}(r, s), !R(r) \rightarrow \text{Fail} \\
\text{Fixed} & \quad \{ \text{send}(s), !S(s), R(r) \rightarrow S(s), \text{Unsafe}(r, s), \text{Fixed} \\
& \quad \text{send}(s), R(r) \rightarrow \text{Unsafe}(r, s), S(s) \\
\text{S(s)} & \quad \{ \text{ack}(r, s'), !R(r), s \neq s' \rightarrow \text{Unsafe}(r, s), S(s) \} \\
\text{Unsafe}(r, s) & \quad \{ \text{send}(s) \rightarrow \text{Fail} \} \\
\text{R(r)} & \quad \{ \text{ack}(r, s) \rightarrow \text{empty} \}
\end{align*}
\]
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.

\[
\begin{align*}
\text{r}_1 & \left\{ \begin{array}{l}
\text{ack}(r, s), \neg \text{r}_1(r, s), \neg \text{r}_2(r, s), \neg \text{r}_2(s), \neg \text{r}_3(s) \rightarrow \text{r}_1, \text{r}_1(r, s) \\
\text{send}(s), \neg \text{r}_1(r, s), \neg \text{r}_2(r, s), \neg \text{r}_2(s), \neg \text{r}_3(s) \rightarrow \text{r}_1, \text{r}_2(r, s)
\end{array} \right. \\
\text{r}_1(r, s) & \left\{ \begin{array}{l}
\text{send}(s) \rightarrow \text{r}_2(r, s)
\end{array} \right. \\
\text{r}_2(s) & \left\{ \begin{array}{l}
\text{send}(s) \rightarrow \text{r}_3(s) \\
\text{ack}(r, s_p), s \neq s_p, \neg \text{r}_1(r, s), \neg \text{r}_2(r, s), \neg \text{r}_3(s) \rightarrow \text{r}_2(s), \text{r}_2(r, s)
\end{array} \right. \\
\text{r}_2(r, s) & \left\{ \begin{array}{l}
\text{send}(s) \rightarrow \text{r}_3(r, s) \\
\text{ack}(r, s) \rightarrow \text{r}_1(r, s)
\end{array} \right. \\
\text{r}_3(s) & \left\{ \begin{array}{l}
\text{ack}(r, s_p), s \neq s_p, \neg \text{r}_1(r, s), \neg \text{r}_2(r, s), \neg \text{r}_3(s) \rightarrow \text{r}_3(s), \text{r}_3(r, s)
\end{array} \right. \\
\text{r}_3(r, s) & \{
\end{align*}
\]

Hand-written version from before we started the process:

\[
\begin{align*}
\text{Start} & \left\{ \begin{array}{l}
\text{send}(s), \neg \text{S}(s) \rightarrow \text{S}(s), \text{Start}
\end{array} \right. \\
\text{Fixed} & \{
\text{ack}(r, s), \neg \text{R}(r) \rightarrow \text{R}(r), \text{Start}
\end{array} \right.
\end{align*}
\]
Properties

Theorem

Given a domain-explicit $Q$, let $RS$ be the rule system given by the above translation. For monitoring state $M_\tau$ and rule state $\Delta_\tau$ if

$$M_\tau = \tau \ast \emptyset \mapsto \{q_0, \sigma_0(Y)\}$$

and

$$\{q_0, \sigma_0\} \xrightarrow{\tau} \Delta_\tau$$

then for any valuation $\theta$

$$M_\tau(\theta) = \{q, \sigma | \langle q, \theta \cup \sigma \cup \sigma' \rangle \in \Delta_\tau \land \text{dom}(\sigma') \cap Y = \emptyset\}$$

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 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.

\[ \forall v \exists p \forall c \]

\[ \text{member}(v, p) \rightarrow \text{candidate}(c, p) \rightarrow \text{rank}(v, c, r) \]

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?