Performance issues and optimizations

Giles Reger

University of Manchester

September 24, 2016

Three Parts

- Optimising Parametric Trace Slicing
- Static Partial Evaluation of Monitors
- Evaluating Runtime Monitoring Tools

Optimising Parametric Trace Slicing

In this part we will consider:

- Extensions to the *expressiveness* of the theory
- Indexing techniques to improve *efficiency*
- Notions of *redundancy* that reduce the work required
- Other pragmatic issues.

Optimising Parametric Trace Slicing

In this part we will consider:

- Extensions to the *expressiveness* of the theory
- Indexing techniques to improve *efficiency*
- Notions of *redundancy* that reduce the work required
- Other pragmatic issues.

Expressiveness: What are the limitations?

How do we use the slicing technique to capture such properties?

- Every counter strictly increases
- Every item on an auction site sells for the maximum of its bids
- Every account has two distinct account managers
- There exists a control tower in each region that, in the last 20 minutes, has communicated with every plane in that region
- For every publisher there exists a subscriber that acknowledges every message the publisher sends

Expressiveness: What are the limitations?

How do we use the slicing technique to capture such properties?

- Every counter strictly increases
- Every item on an auction site sells for the maximum of its bids
- Every account has two distinct account managers
- There exists a control tower in each region that, in the last 20 minutes, has communicated with every plane in that region
- For every publisher there exists a subscriber that acknowledges every message the publisher sends

Some of these:

- Require data to be processed *locally* to each slice

Expressiveness: What are the limitations?

How do we use the slicing technique to capture such properties?

- Every counter strictly increases
- Every item on an auction site sells for the maximum of its bids
- Every account has two distinct account managers
- There exists a control tower in each region that, in the last 20 minutes, has communicated with every plane in that region
- For every publisher there exists a subscriber that acknowledges every message the publisher sends

Some of these:

- Require data to be processed *locally* to each slice
- Require the results of slices to be combined non-universally

Data Local Processing

- Let us take the property: *Every counter strictly increases*
- We observe the event counter(*id*,*value*)
- The property is *for every counter* so we slice on counter *ids*
- For example, the trace

  $$\texttt{counter}(A,2).\texttt{counter}(B,5).\texttt{counter}(A,3).\texttt{counter}(B,5)$$

  has two slices (for A and B) with the one for B being 'wrong'
- Without keeping the *value* data values in the projected trace we cannot tell this
- Therefore the solution for data local processing is
  1. Define projection to preserve parameters
  2. Define *plugin* languages over parameterised traces

Data Local Processing

- Let us take the property: *Every counter strictly increases*
- We observe the event counter(*id*,*value*)
- The property is *for every counter* so we slice on counter *ids*
- For example, the trace

  $\texttt{counter}(A, 2).\texttt{counter}(B, 5).\texttt{counter}(A, 3).\texttt{counter}(B, 5)$

  has two slices (for A and B) with the one for B being 'wrong'
- Without keeping the *value* data values in the projected trace we cannot tell this
- Therefore the solution for data local processing is
  1. Define projection to preserve parameters
  2. Define *plugin* languages over parameterised traces

Data Local Processing

- Let us take the property: *Every counter strictly increases*
- We observe the event counter(*id*,*value*)
- The property is *for every counter* so we slice on counter *ids*
- For example, the trace

  $\texttt{counter}(A, 2).\texttt{counter}(B, 5).\texttt{counter}(A, 3).\texttt{counter}(B, 5)$

  has two slices (for A and B) with the one for B being 'wrong'
- Without keeping the *value* data values in the projected trace we cannot tell this
- Therefore the solution for data local processing is
  1. Define projection to preserve parameters
  2. Define *plugin* languages over parameterised traces

Data Local Processing

- Let us take the property: *Every counter strictly increases*
- We observe the event `counter`(*id*,*value*)
- The property is *for every counter* so we slice on counter *ids*
- For example, the trace

  $$\texttt{counter}(A, 2).\texttt{counter}(B, 5).\texttt{counter}(A, 3).\texttt{counter}(B, 5)$$

  has two slices (for A and B) with the one for B being 'wrong'
- Without keeping the *value* data values in the projected trace we cannot tell this
- Therefore the solution for data local processing is
  1. Define projection to preserve parameters
  2. Define *plugin* languages over parameterised traces

On the Relation between Concrete and Abstract Events

- A small note....
- Take the property: *No two counters have the same value*
- With the same observed event `counter`(id,$value$)
- Now we need to talk about two counters. So we really need two events

$$\texttt{counter}(id_1, value) \qquad \texttt{counter}(id_2, value)$$

- This is easily supported by the slicing theory (e.g. in `tracematches`). But the work of JavaMOP assumes an implicit mapping between event names and parameters
- There is, of course, the case where $id_1 = id_2$ to deal with

Non-Universal Acceptance

- Let us take the property: *Every account has two distinct account managers*
- We observe the event `isManager`(*account*,*manager*)
- The property says that *for every* account *a there exists* managers m_1 and m_2 such that $m_1 \neq m_2$ and eventually `isManager`(a,m_1) and `isManager`(a,m_2)

- We cannot capture the property by defining a property that must hold for every account and manager
- Or even every account and pair of managers
- We need to write $\forall a \exists m_1 \exists m_2 : m_1 \neq m_2 \wedge \varphi$ (or similar)

One Solution: Quantified Event Automata

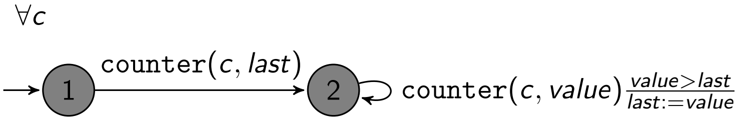
Quantified event automata (QEA) (see Barringer 2012) is a slicing-baesd formalism that solves all of the above issues. It has:

- A plugin language over parameterised traces (event automata) that are extended finite state machines with guards and assignments on transitions
- A general alphabet (i.e. no implicit mapping)
- Arbitrary quantification (including empty) with guards

There exists a tool called MarQ (Monitoring At Runtime with QEA) for monitoring specifications written as QEAs.

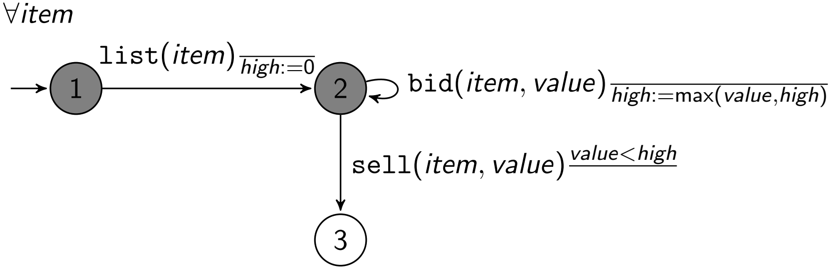
Brief Examples

Every counter strictly increases



Brief Examples

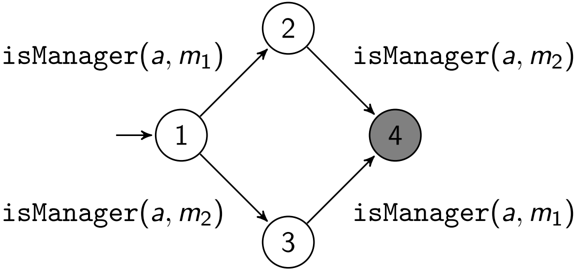
Every item on an auction site sells for the maximum of its bids



Brief Examples

Every account has two distinct account managers

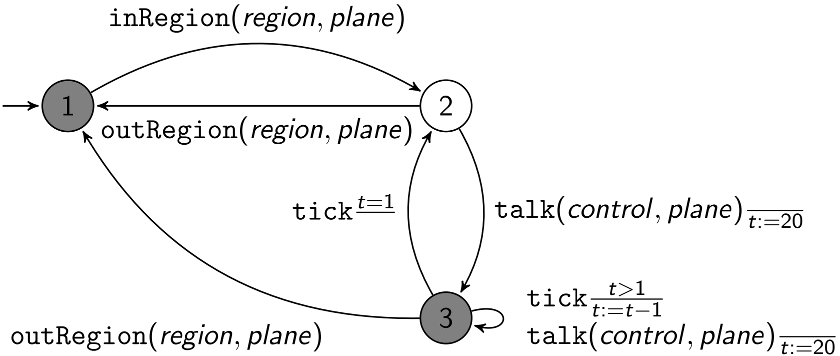
$\forall a \exists m_1 \exists m_2 : m_1 \neq m_2$



Brief Examples

There exists a control tower in each region that, in the last 20 minutes, has communicated with every plane in that region

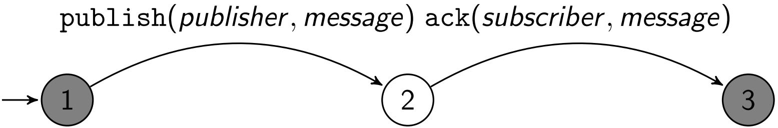
$\forall region \exists control \forall plane$



Brief Examples

For every publisher there exists a subscriber that acknowledges every message the publisher sends

$\forall publisher \exists subscriber \forall message$



On Algorithms

- These changes affect how the algorithms discussed in this and the previous lecture behave
- The two main differences come from
  - Dealing with the general alphabet, especially the case where two symbolic events match the same concrete event
  - Dealing with *free* (unquantified) variables (and guards and assignments)

- For time/space reasons we will not discuss QEAs further
- In the next part we will assume the previous semantics

Optimising Parametric Trace Slicing

In this part we will consider:

- Extensions to the *expressiveness* of the theory
- Indexing techniques to improve *efficiency*
- Notions of *redundancy* that reduce the work required
- Other pragmatic issues.

Previously...

We saw an algorithm for monitoring JavaMOP properties...

$\Delta : [Bind \rightharpoonup State]$; $\Theta : Bind$;
$\Delta \leftarrow [\bot \rightarrow q_0]$;
foreach $e(\theta) \in \tau$ *in order* **do**
  $\Theta \leftarrow \underline{dom}(\Delta)$;
  foreach $\theta' \in \Theta$ **do**
    if θ *is consistent with* θ' **then**
      $\theta_{max} \leftarrow \theta'$;
      foreach $\theta_{alt} \in \Theta$ **do**
        if $\theta_{max} \sqsubseteq \theta_{alt} \sqsubseteq \theta \dagger \theta'$ **then** $\theta_{max} = \theta_{alt}$;
      $\Delta(\theta \dagger \theta') \leftarrow \delta(\Delta(\theta_{max}), e)$
return $\theta \in \underline{dom}(\Delta)$ where $\Delta(\theta)$ is final

Previously...

We saw an algorithm for monitoring JAVAMOP properties...

$\Delta : [Bind \rightharpoonup State]$; $\Theta : Bind$;
$\Delta \leftarrow [\bot \rightarrow q_0]$;
foreach $e(\theta) \in \tau$ *in order* **do**
  $\Theta \leftarrow \underline{dom}(\Delta)$;
  foreach $\theta' \in \Theta$ **do**
    if θ *is consistent with* θ' **then**
      $\theta_{max} \leftarrow \theta'$;
      foreach $\theta_{alt} \in \Theta$ **do**
        if $\theta_{max} \sqsubseteq \theta_{alt} \sqsubseteq \theta \dagger \theta'$ **then** $\theta_{max} = \theta_{alt}$;
      $\Delta(\theta \dagger \theta') \leftarrow \delta(\Delta(\theta_{max}), e)$
return $\theta \in \underline{dom}(\Delta)$ where $\Delta(\theta)$ is final

- Let $n = |\underline{dom}(\Delta)|$ on a given step
- There are n^2 accesses to Δ for each event

Value-Based Indexing

- The reason for the n^2 accesses is that we check every binding to see if it is relevant to the event
- This is clearly inefficient

- Instead, we can directly lookup relevant events by storing in a map, for each binding, those existing bindings that are relevant

- This is called value-based indexing as we are indexing on the values (parameters) of the event

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \rightarrow 2^{\mathit{Bind}}$ be such a map

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \rightarrow 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \rightarrow 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ
- Δ should be 'union-closed' - if two compatible bindings are in Δ, their union should also be in Δ:

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \rightarrow 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ
- Δ should be ‘union-closed’ - if two compatible bindings are in Δ, their union should also be in Δ:

  This ensures that the most informative bindings are in Δ

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \rightarrow 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ
- Δ should be 'union-closed' - if two compatible bindings are in Δ, their union should also be in Δ:

$$\forall \theta, \theta' \in \underline{\mathit{dom}}(\Delta) : \mathit{compatible}(\theta, \theta') \Rightarrow \theta \sqcup \theta' \in \underline{\mathit{dom}}(\Delta)$$

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \rightarrow 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ
- Δ should be 'union-closed' - if two compatible bindings are in Δ, their union should also be in Δ:

$$\forall \theta, \theta' \in \underline{\mathit{dom}}(\Delta) : \mathit{compatible}(\theta, \theta') \Rightarrow \theta \sqcup \theta' \in \underline{\mathit{dom}}(\Delta)$$

- $\mathbb{U}$ should be 'submap-closed' - every submap of a binding in Δ should be in $\mathbb{U}$:

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \rightarrow 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ
- Δ should be 'union-closed' - if two compatible bindings are in Δ, their union should also be in Δ:

$$\forall \theta, \theta' \in \underline{\mathit{dom}}(\Delta) : \mathit{compatible}(\theta, \theta') \Rightarrow \theta \sqcup \theta' \in \underline{\mathit{dom}}(\Delta)$$

- $\mathbb{U}$ should be 'submap-closed' - every submap of a binding in Δ should be in $\mathbb{U}$:

This ensures that every partial binding will be related to the known larger bindings

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \to 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ
- Δ should be 'union-closed' - if two compatible bindings are in Δ, their union should also be in Δ:

$$\forall \theta, \theta' \in \underline{\mathit{dom}}(\Delta) : \mathit{compatible}(\theta, \theta') \Rightarrow \theta \sqcup \theta' \in \underline{\mathit{dom}}(\Delta)$$

- $\mathbb{U}$ should be 'submap-closed' - every submap of a binding in Δ should be in $\mathbb{U}$:

$$\forall \theta \in \underline{\mathit{dom}}(\Delta), \forall \theta' \in \mathit{Bind} : \theta' \sqsubset \theta \Rightarrow \theta' \in \underline{\mathit{dom}}(\mathbb{U})$$

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \rightarrow 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ
- Δ should be 'union-closed' - if two compatible bindings are in Δ, their union should also be in Δ:

$$\forall \theta, \theta' \in \underline{\mathit{dom}}(\Delta) : \mathit{compatible}(\theta, \theta') \Rightarrow \theta \sqcup \theta' \in \underline{\mathit{dom}}(\Delta)$$

- $\mathbb{U}$ should be 'submap-closed' - every submap of a binding in Δ should be in $\mathbb{U}$:

$$\forall \theta \in \underline{\mathit{dom}}(\Delta), \forall \theta' \in \mathit{Bind} : \theta' \sqsubset \theta \Rightarrow \theta' \in \underline{\mathit{dom}}(\mathbb{U})$$

- $\mathbb{U}$ should be 'relevance-closed' - every entry in $\mathbb{U}$ should point to the relevant bindings in Δ:

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \rightarrow 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ
- Δ should be 'union-closed' - if two compatible bindings are in Δ, their union should also be in Δ:

$$\forall \theta, \theta' \in \underline{\mathit{dom}}(\Delta) : \mathit{compatible}(\theta, \theta') \Rightarrow \theta \sqcup \theta' \in \underline{\mathit{dom}}(\Delta)$$

- $\mathbb{U}$ should be 'submap-closed' - every submap of a binding in Δ should be in $\mathbb{U}$:

$$\forall \theta \in \underline{\mathit{dom}}(\Delta), \forall \theta' \in \mathit{Bind} : \theta' \sqsubset \theta \Rightarrow \theta' \in \underline{\mathit{dom}}(\mathbb{U})$$

- $\mathbb{U}$ should be 'relevance-closed' - every entry in $\mathbb{U}$ should point to the relevant bindings in Δ:

This is the point of $\mathbb{U}$...to point to the relevant known bindings

What should $\mathbb{U}$ be?

- Let $\mathbb{U} : \mathit{Bind} \to 2^{\mathit{Bind}}$ be such a map
- We want $\mathbb{U}$ to help us update Δ
- Δ should be 'union-closed' - if two compatible bindings are in Δ, their union should also be in Δ:

$$\forall \theta, \theta' \in \underline{\mathit{dom}}(\Delta) : \mathit{compatible}(\theta, \theta') \Rightarrow \theta \sqcup \theta' \in \underline{\mathit{dom}}(\Delta)$$

- $\mathbb{U}$ should be 'submap-closed' - every submap of a binding in Δ should be in $\mathbb{U}$:

$$\forall \theta \in \underline{\mathit{dom}}(\Delta), \forall \theta' \in \mathit{Bind} : \theta' \sqsubset \theta \Rightarrow \theta' \in \underline{\mathit{dom}}(\mathbb{U})$$

- $\mathbb{U}$ should be 'relevance-closed' - every entry in $\mathbb{U}$ should point to the relevant bindings in Δ:

$$\forall \theta, \theta' \in \underline{\mathit{dom}}(\Delta) : \theta \sqsubseteq \theta' \Rightarrow \theta' \in \mathbb{U}(\theta)$$

A refined algorithm

$\Delta : [Bind \rightharpoonup State]; \mathbb{U} : Bind \rightarrow 2^{Bind}$
$\Delta \leftarrow \{\bot \rightarrow q_0\}; \mathbb{U} \leftarrow \emptyset$ for any $\theta \in Bind$
foreach $e(\theta) \in \tau$ *in order* **do**
  if $\theta \notin \underline{dom}(\Delta)$ **then**
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
      if $\theta_m \in \underline{dom}(\Delta)$ **then** break;
    $\texttt{defTo}(\theta, \theta_m)$
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
      foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
        if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then**
        $\texttt{defTo}(\theta' \sqcup \theta, \theta')$;
  foreach $\theta' \in \{\theta\} \cup \mathbb{U}(\theta)$ **do**
    $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
return Δ

A refined algorithm

$\Delta : [\mathit{Bind} \rightharpoonup \mathit{State}]; \mathbb{U} : \mathit{Bind} \rightarrow 2^{\mathit{Bind}}$

$\Delta \leftarrow \{\bot \rightarrow q_0\}; \mathbb{U} \leftarrow \emptyset$ for any $\theta \in \mathit{Bind}$

foreach $e(\theta) \in \tau$ *in order* **do**

  if $\theta \notin \underline{dom}(\Delta)$ **then**

    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**

      if $\theta_m \in \underline{dom}(\Delta)$ **then** break;

    `defTo`(θ, θ_m)

    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**

      foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**

        if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then** `defTo`$(\theta' \sqcup \theta, \theta')$;

  foreach $\theta' \in \{\theta\} \cup \mathbb{U}(\theta)$ **do**

    $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$

return Δ

- Initialisation

A refined algorithm

$\Delta : [Bind \rightharpoonup State]; \mathbb{U} : Bind \to 2^{Bind}$
$\Delta \leftarrow \{\bot \to q_0\}; \mathbb{U} \leftarrow \emptyset$ for any $\theta \in Bind$
foreach $e(\theta) \in \tau$ *in order* **do**
  if $\theta \notin \underline{dom}(\Delta)$ **then**
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
      if $\theta_m \in \underline{dom}(\Delta)$ **then** break;
    $\texttt{defTo}(\theta, \theta_m)$
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
      foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
        if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then** $\texttt{defTo}(\theta' \sqcup \theta, \theta')$;
  foreach $\theta' \in \{\theta\} \cup \mathbb{U}(\theta)$ **do**
    $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
return Δ

- Initialisation
- For each event

A refined algorithm

$\Delta : [Bind \rightharpoonup State]; \mathbb{U} : Bind \rightarrow 2^{Bind}$

$\Delta \leftarrow \{\bot \rightarrow q_0\}; \mathbb{U} \leftarrow \emptyset$ for any $\theta \in Bind$

foreach $e(\theta) \in \tau$ *in order* **do**

  if $\theta \notin \underline{dom}(\Delta)$ **then**

    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**

      if $\theta_m \in \underline{dom}(\Delta)$ **then** break;

    $\texttt{defTo}(\theta, \theta_m)$

    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**

      foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**

        if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then** $\texttt{defTo}(\theta' \sqcup \theta, \theta')$;

  foreach $\theta' \in \{\theta\} \cup \mathbb{U}(\theta)$ **do**

    $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$

return Δ

- Initialisation
- For each event
- If θ is not defined add it and ensure closure properties

We will look at how this is done next

A refined algorithm

$\Delta : [Bind \rightharpoonup State]; \mathbb{U} : Bind \rightarrow 2^{Bind}$

$\Delta \leftarrow \{\bot \rightarrow q_0\}; \mathbb{U} \leftarrow \emptyset$ for any $\theta \in Bind$

foreach $e(\theta) \in \tau$ *in order* **do**

    if $\theta \notin \underline{dom}(\Delta)$ **then**

        foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**

            if $\theta_m \in \underline{dom}(\Delta)$ **then** break;

        `defTo`(θ, θ_m)

        foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**

            foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**

                if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then** `defTo`$(\theta' \sqcup \theta, \theta')$;

    foreach $\theta' \in \{\theta\} \cup \mathbb{U}(\theta)$ **do**

        $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$

return Δ

- Initialisation
- For each event
- If θ is not defined add it and ensure closure properties

  We will look at how this is done next
- Update states for relevant bindings

Closing $\mathbb{U}$

if $\theta \notin \underline{dom}(\Delta)$ **then**
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
        if $\theta_m \in \underline{dom}(\Delta)$ **then** break ;
    `defTo`(θ, θ_m)
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
        foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
            if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then** `defTo`$(\theta' \sqcup \theta, \theta')$;

- We only need to update $\mathbb{U}$ if θ is not in $\mathbb{U}$

Closing $\mathbb{U}$

if $\theta \notin \underline{dom}(\Delta)$ **then**
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
        if $\theta_m \in \underline{dom}(\Delta)$ **then** break ;
    `defTo`(θ, θ_m)
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
        foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
            if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then** `defTo`$(\theta' \sqcup \theta, \theta')$;

- We only need to update $\mathbb{U}$ if θ is not in $\mathbb{U}$
- We first find the maximal binding in Δ (might be $\bot$)

Closing $\mathbb{U}$

if $\theta \notin \underline{dom}(\Delta)$ **then**
  foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
    if $\theta_m \in \underline{dom}(\Delta)$ **then** break ;
  $\texttt{defTo}(\theta, \theta_m)$
  foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
    foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
      if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then**
      $\texttt{defTo}(\theta' \sqcup \theta, \theta')$;

- We only need to update $\mathbb{U}$ if θ is not in $\mathbb{U}$
- We first find the maximal binding in Δ (might be $\bot$)
- Use it to add θ

Closing $\mathbb{U}$

```
if θ ∉ dom(Δ) then
    foreach θ_m ⊏ θ (big to small) do
        if θ_m ∈ dom(Δ) then break ;
    defTo(θ, θ_m)
    foreach θ_m ⊏ θ (big to small) do
        foreach θ' ∈ U(θ_m) compatible with θ do
            if (θ' ⊔ θ) ∉ dom(Δ) then
                defTo(θ' ⊔ θ, θ');
```

- We only need to update $\mathbb{U}$ if θ is not in $\mathbb{U}$
- We first find the maximal binding in Δ (might be $\bot$)
- Use it to add θ
- Ensures closure properties

Closing $\mathbb{U}$

if $\theta \notin \underline{dom}(\Delta)$ **then**
  foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
    if $\theta_m \in \underline{dom}(\Delta)$ **then** break ;
  defTo(θ, θ_m)
  foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
    foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
      if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then** defTo$(\theta' \sqcup \theta, \theta')$;

- We only need to update $\mathbb{U}$ if θ is not in $\mathbb{U}$
- We first find the maximal binding in Δ (might be $\perp$)
- Use it to add θ
- Ensures closure properties
- Consider all submaps

Closing $\mathbb{U}$

if $\theta \notin \underline{dom}(\Delta)$ **then**
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
        if $\theta_m \in \underline{dom}(\Delta)$ **then** break ;
    `defTo`(θ, θ_m)
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
        foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
            if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then** `defTo`$(\theta' \sqcup \theta, \theta')$;

- We only need to update $\mathbb{U}$ if θ is not in $\mathbb{U}$
- We first find the maximal binding in Δ (might be $\bot$)
- Use it to add θ
- Ensures closure properties
- Consider all submaps
- Attempt to create all unions

Closing $\mathbb{U}$

if $\theta \notin \underline{dom}(\Delta)$ **then**
  foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
    if $\theta_m \in \underline{dom}(\Delta)$ **then** break ;
  $\texttt{defTo}(\theta, \theta_m)$
  foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
    foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
      if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then**
        $\texttt{defTo}(\theta' \sqcup \theta, \theta')$;

...
$\texttt{defTo}(\theta, \theta')$:
  $\Delta(\theta) \leftarrow \Delta(\theta')$
  foreach $\theta'' \sqsubset \theta$ **do**
    $\mathbb{U}(\theta'') \leftarrow \mathbb{U}(\theta'') \cup \{\theta\}$;

- We only need to update $\mathbb{U}$ if θ is not in $\mathbb{U}$
- We first find the maximal binding in Δ (might be $\perp$)
- Use it to add θ
- Ensures closure properties
- Consider all submaps
- Attempt to create all unions
- defTo

Closing $\mathbb{U}$

if $\theta \notin \underline{dom}(\Delta)$ **then**
  foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
    if $\theta_m \in \underline{dom}(\Delta)$ **then** break ;
  defTo(θ, θ_m)
  foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
    foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
      if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then**
      defTo$(\theta' \sqcup \theta, \theta')$;

...

defTo(θ, θ'):
  $\Delta(\theta) \leftarrow \Delta(\theta')$
  foreach $\theta'' \sqsubset \theta$ **do**
    $\mathbb{U}(\theta'') \leftarrow \mathbb{U}(\theta'') \cup \{\theta\}$;

- We only need to update $\mathbb{U}$ if θ is not in $\mathbb{U}$
- We first find the maximal binding in Δ (might be $\bot$)
- Use it to add θ
- Ensures closure properties
- Consider all submaps
- Attempt to create all unions
- `defTo` uses the state from the maximal binding to initialise θ

Closing $\mathbb{U}$

if $\theta \notin \underline{dom}(\Delta)$ **then**
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
        if $\theta_m \in \underline{dom}(\Delta)$ **then** break ;
    defTo(θ, θ_m)
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
        foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
            if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then**
            defTo$(\theta' \sqcup \theta, \theta')$;

...
defTo(θ, θ'):
  $\Delta(\theta) \leftarrow \Delta(\theta')$
  foreach $\theta'' \sqsubset \theta$ **do**
    $\mathbb{U}(\theta'') \leftarrow \mathbb{U}(\theta'') \cup \{\theta\}$;

- We only need to update $\mathbb{U}$ if θ is not in $\mathbb{U}$
- We first find the maximal binding in Δ (might be $\perp$)
- Use it to add θ
- Ensures closure properties
- Consider all submaps
- Attempt to create all unions
- defTo uses the state from the maximal binding to initialise θ
- Relevance-closes $\mathbb{U}$ for θ i.e. adds it to the $\mathbb{U}$-entry for all smaller existing bindings

Why is this better?

```
foreach e(θ) ∈ τ in order do
    if θ ∉ dom(Δ) then
        foreach θ_m ⊏ θ (big to small) do
            if θ_m ∈ dom(Δ) then break;
        defTo(θ, θ_m)
        foreach θ_m ⊏ θ (big to small) do
            foreach θ′ ∈ 𝕌(θ_m) compatible with θ do
                if (θ′ ⊔ θ) ∉ dom(Δ) then defTo(θ′ ⊔ θ, θ′);
    foreach θ′ ∈ {θ} ∪ 𝕌(θ) do
        Δ(θ′) ← σ(Δ(θ′), e)
return Δ
```

- We only update $\mathbb{U}$ if we haven't seen the event's objects before.

```
defTo(θ, θ′):
  Δ(θ) ← Δ(θ′)
  foreach θ″ ⊏ θ do
    𝕌(θ″) ← 𝕌(θ″) ∪ {θ};
```

Why is this better?

foreach $e(\theta) \in \tau$ *in order* **do**
  if $\theta \notin \underline{dom}(\Delta)$ **then**
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
      if $\theta_m \in \underline{dom}(\Delta)$ **then** break;
    $\texttt{defTo}(\theta, \theta_m)$
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
      foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
        if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then** $\texttt{defTo}(\theta' \sqcup \theta, \theta')$;
  foreach $\theta' \in \{\theta\} \cup \mathbb{U}(\theta)$ **do**
    $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
return Δ

- We only update $\mathbb{U}$ if we haven't seen the event's objects before.

  Optimise Common Case

$\texttt{defTo}(\theta, \theta')$:
  $\Delta(\theta) \leftarrow \Delta(\theta')$
  foreach $\theta'' \sqsubset \theta$ **do**
    $\mathbb{U}(\theta'') \leftarrow \mathbb{U}(\theta'') \cup \{\theta\}$;

Why is this better?

foreach $e(\theta) \in \tau$ *in order* **do**
  if $\theta \notin \underline{dom}(\Delta)$ **then**
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
      if $\theta_m \in \underline{dom}(\Delta)$ **then** break;
    $\texttt{defTo}(\theta, \theta_m)$
    foreach $\theta_m \sqsubset \theta$ *(big to small)* **do**
      foreach $\theta' \in \mathbb{U}(\theta_m)$ *compatible with* θ **do**
        if $(\theta' \sqcup \theta) \notin \underline{dom}(\Delta)$ **then** $\texttt{defTo}(\theta' \sqcup \theta, \theta')$;
  foreach $\theta' \in \{\theta\} \cup \mathbb{U}(\theta)$ **do**
    $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
return Δ

- We only update $\mathbb{U}$ if we haven't seen the event's objects before.

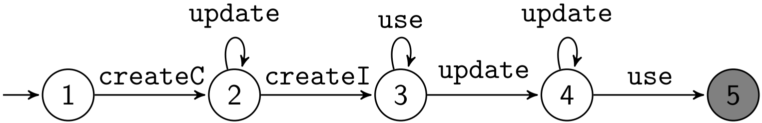
Optimise Common Case

- Only iterate over small collections - we expect $\mathbb{U}(\theta)$ to be small compared to $\underline{dom}(\Delta)$.

$\texttt{defTo}(\theta, \theta')$:
  $\Delta(\theta) \leftarrow \Delta(\theta')$
  foreach $\theta'' \sqsubset \theta$ **do** $\mathbb{U}(\theta'') \leftarrow \mathbb{U}(\theta'') \cup \{\theta\}$;

How it works

Recall the UnsafeMapIterator example used previously.



createC(M_1,C_1)
createC(M_1,C_2)
createI(C_1,I_1)
update(C_1)
createI(C_2,I_2)
use(I_1)

How it works

We begin with Δ containing the empty binding and initial state, and $\mathbb{U}$ empty

Trace	Δ		$\mathbb{U}$
	(-,-,-)	1	

How it works

Adding (M_1,-,-) and (-,C_1,-) to $\mathbb{U}$ allows us to find (M_1,C_1,-) in the future whenever we see an event using just C_1 or M_1

Trace	Δ		$\mathbb{U}$	
	(-,-,-)	1	(-,-,-)	(M_1,C_1,-)
createC(M_1,C_1)	(M_1,C_1,-)	2		
			(M_1,-,-)	(M_1,C_1,-)
			(-,C_1,-)	(M_1,C_1,-)

How it works

(M_1,C_2,-) is also added to the entry in $\mathbb{U}$ for (M_1,-,-) - this relates to the 'above-of' relation in the lattice we were building earlier

Trace	Δ		$\mathbb{U}$	
	(-,-,-)	1	(-,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createC(M_1,C_1)	(M_1,C_1,-)	2		
createC(M_1,C_2)	(M_1,C_2,-)	2		
			(M_1,-,-)	(M_1,C_1,-)(M_1,C_2,-)
			(-,C_1,-)	(M_1,C_1,-)
			(-,C_2,-)	(M_1,C_2,-)

How it works

$(\text{-},C_1,I_1)$ is added from $(\text{-},\text{-},\text{-})$ $(M_1,C_1,\text{-})$ in $\mathbb{U}((\text{-},C_1,\text{-}))$ is used to add (M_1,C_1,I_1)

Trace	Δ	
	$(\text{-},\text{-},\text{-})$	1
createC(M_1,C_1)	$(M_1,C_1,\text{-})$	2
createC(M_1,C_2)	$(M_1,C_2,\text{-})$	2
createI(C_1,I_1)	$(\text{-},C_1,I_1)$	F
	(M_1,C_1,I_1)	3

$\mathbb{U}$	
$(\text{-},\text{-},\text{-})$	$(M_1,C_1,\text{-})(M_1,C_2,\text{-})$
	$(\text{-},C_1,I_1)(M_1,C_1,I_1)$
$(M_1,\text{-},\text{-})$	$(M_1,C_1,\text{-})(M_1,C_2,\text{-})$
	(M_1,C_1,I_1)
$(\text{-},C_1,\text{-})$	$(M_1,C_1,\text{-})(\text{-},C_1,I_1)$
	(M_1,C_1,I_1)
$(\text{-},C_2,\text{-})$	$(M_1,C_2,\text{-})(\text{-},C_2,I_2)$
$(\text{-},\text{-},I_1)$	$(\text{-},C_1,I_1)(M_1,C_1,I_1)$
$(M_1,C_1,\text{-})$	(M_1,C_1,I_1)
$(\text{-},C_1,I_1)$	(M_1,C_1,I_1)
$(M_1,\text{-},I_1)$	(M_1,C_1,I_1)

How it works

θ_m is (-,-,-) therefore `defTo`((-,C_1,-),(-,-,-)) sets Δ((-,C_1,-))=1 which is updated to F. As expected $\mathbb{U}$((-,C_1,-)) = {(M_1,C_1,-),(-,C_1,I_1),(M_1,C_1,I_1)}

Trace
`createC`(M_1,C_1)
`createC`(M_1,C_2)
`createI`(C_1,I_1)
`update`(C_1)

Δ	
(-,-,-)	1
(M_1,C_1,-)	2
(M_1,C_2,-)	2
(-,C_1,I_1)	F
(M_1,C_1,I_1)	4
(-,C_1,-)	F

$\mathbb{U}$	
(-,-,-)	(M_1,C_1,-)(M_1,C_2,-) (-,C_1,I_1)(M_1,C_1,I_1) (-,C_1,-)
(M_1,-,-)	(M_1,C_1,-)(M_1,C_2,-) (M_1,C_1,I_1)
(-,C_1,-)	(M_1,C_1,-)(-,C_1,I_1) (M_1,C_1,I_1)
(-,C_2,-)	(M_1,C_2,-)
(-,-,I_1)	(-,C_1,I_1)(M_1,C_1,I_1)
(M_1,C_1,-)	(M_1,C_1,I_1)
(-,C_1,I_1)	(M_1,C_1,I_1)
(M_1,-,I_1)	(M_1,C_1,I_1)

How it works

We consider $(\text{-},C_2,\text{-}) \sqsubset (\text{-},C_2,I_2)$ and use $\mathbb{U}((\text{-},C_2,\text{-}))$ to add (M_1,C_2,I_2)

Trace
createC(M_1,C_1)
createC(M_1,C_2)
createI(C_1,I_1)
update(C_1)
createI(C_2,I_2)

Δ	
$(\text{-},\text{-},\text{-})$	1
$(M_1,C_1,\text{-})$	2
$(M_1,C_2,\text{-})$	2
$(\text{-},C_1,I_1)$	F
(M_1,C_1,I_1)	4
$(\text{-},C_1,\text{-})$	F
$(\text{-},C_2,I_2)$	F
(M_1,C_2,I_2)	3

$\mathbb{U}$	
$(\text{-},\text{-},\text{-})$	$(M_1,C_1,\text{-})(M_1,C_2,\text{-})$ $(\text{-},C_1,I_1)(M_1,C_1,I_1)$ $(\text{-},C_1,\text{-})(\text{-},C_2,I_2)$ (M_1,C_2,I_2)
$(M_1,\text{-},\text{-})$	$(M_1,C_1,\text{-})(M_1,C_2,\text{-})$ $(M_1,C_1,I_1)(M_1,C_2,I_2)$
$(\text{-},C_1,\text{-})$	$(M_1,C_1,\text{-})(\text{-},C_1,I_1)$ (M_1,C_1,I_1)
$(\text{-},C_2,\text{-})$	$(M_1,C_2,\text{-})(\text{-},C_2,I_2)$ (M_1,C_2,I_2)
...	...
$(\text{-},\text{-},I_2)$	$(\text{-},C_2,I_2)(M_1,C_2,I_2)$
$(M_1,C_2,\text{-})$	(M_1,C_2,I_2)
$(\text{-},C_2,I_2)$	(M_1,C_2,I_2)
$(M_1,\text{-},I_2)$	(M_1,C_2,I_2)

How it works

We can use the (-,-,I_1) entry in $\mathbb{U}$ to find the two relevant bindings.
Previously we would have had to compare (-,-,I_1) with every binding in Δ

Trace	Δ		$\mathbb{U}$	
	(-,-,-)	1	(-,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createC(M_1,C_1)	(M_1,C_1,-)	2		(-,C_1,I_1)(M_1,C_1,I_1)
createC(M_1,C_2)	(M_1,C_2,-)	2		(-,C_1,-)1-¿(-,C_2,I_2)
createI(C_1,I_1)	(-,C_1,I_1)	F		(M_1,C_2,I_2)(-,-,I_1)
update(C_1)	(M_1,C_1,I_1)	5	(M_1,-,-)	(M_1,C_1,-)(M_1,C_2,-)
createI(C_2,I_2)	(-,C_1,-)	F		(M_1,C_1,I_1)(M_1,C_2,I_2)
use(I_1)	(-,C_2,I_2)	F	(-,C_1,-)	(M_1,C_1,-)(-,C_1,I_1)
	(M_1,C_2,I_2)	3		(M_1,C_1,I_1)
			(-,C_2,-)	(M_1,C_2,-)(-,C_2,I_2)
				(M_1,C_2,I_2)
			(-,-,I_1)	(-,C_1,I_1)(M_1,C_1,I_1)
			(M_1,C_1,-)	(M_1,C_1,I_1)
			...	...

How it works

Trace
`createC`(M_1,C_1)
`createC`(M_1,C_2)
`createI`(C_1,I_1)
`update`(C_1)
`createI`(C_2,I_2)
`use`(I_1)

Δ	
$(-,-,-)$	1
$(M_1,C_1,-)$	F
$(M_1,C_2,-)$	2
$(-,C_1,I_1)$	F
(M_1,C_1,I_1)	5
$(-,C_1,-)$	F
$(-,C_2,I_2)$	F
(M_1,C_2,I_2)	3

$\mathbb{U}$	
$(-,-,-)$	$(M_1,C_1,-)(M_1,C_2,-)$
	$(-,C_1,I_1)(M_1,C_1,I_1)$
	$(-,C_1,-)(-,C_2,I_2)$
	$(M_1,C_2,I_2)(-,-,I_1)$
$(M_1,-,-)$	$(M_1,C_1,-)(M_1,C_2,-)$
	$(M_1,C_1,I_1)(M_1,C_2,I_2)$
$(-,C_1,-)$	$(M_1,C_1,-)(-,C_1,I_1)$
	(M_1,C_1,I_1)
$(-,C_2,-)$	$(M_1,C_2,-)(-,C_2,I_2)$
	(M_1,C_2,I_2)
$(-,-,I_1)$	$(-,C_1,I_1)(M_1,C_1,I_1)$
$(M_1,C_1,-)$	(M_1,C_1,I_1)
$(-,C_1,I_1)$	(M_1,C_1,I_1)
$(M_1,-,I_1)$	(M_1,C_1,I_1)
$(-,-,I_2)$	$(-,C_2,I_2)(M_1,C_2,I_2)$
$(M_1,C_2,-)$	(M_1,C_2,I_2)
$(-,C_2,I_2)$	(M_1,C_2,I_2)
$(M_1,-,I_2)$	(M_1,C_2,I_2)

Other kinds of Indexing

- The idea here was to lookup the relevant bindings using the values in an event
- There are two other possibilities:
  - State-based. Associate states with the bindings in those states (only beneficial in suffix-matching)
  - Symbol-based. Use the event names to find the bindings in states where those events have transitions that cause the binding to change state.
- It is possible to combine the kinds of indexing
  - `tracematches` combines State and Value
  - MarQ combines Symbol and Value

Distributed Indexing

- The idea is to use AspectJ weaving to distribute indexing directly into the relevant objects

- The simple idea: single object indexing
- Instead of having a map relating objects to the relevant states, add that relevant state directly into the object

- For multi-object indexing a *master* object is chosen per parameter list and the index distributed into that object. The details depend on how indexing is organised generally.

- The disadvantages of this approach are
  - Restricted to online monitoring of Java programs using AspectJ
  - The amount of instrumentation significantly increases
  - It may require modifying libraries (e.g. the code of Map)

The Hierarchical Fragment

- The recent work of those behind the MUFIN tool has introduced a new indexing technique
- They noticed that most of the properties used in benchmarks+papers have a certain property that when multiple objects are monitored *one is created from the other*
- This leads to a fragment of the slicing theory (which I call the hierarchical fragment)
- It also leads to a (very) efficient indexing technique that organises everything in terms of this hierarchy. Briefly,
  - Monitored objects are extended to point to the monitored objects below them in the hierarchy
  - These objects are organised into sets according to the state the combination of objects is in
  - This allows monitoring steps to be implemented using union-find techniques

Optimising Parametric Trace Slicing

In this part we will consider:

- Extensions to the *expressiveness* of the theory
- Indexing techniques to improve *efficiency*
- Notions of *redundancy* that reduce the work required
- Other pragmatic issues.

What is Redundant?

- Looking at the algorithm we have so far, where can we find *redundancies*?

What is Redundant?

- Looking at the algorithm we have so far, where can we find *redundancies*?

- We process each *event*
- With respect to existing *bindings*

- Work is proportional to the number of each

- We want to find when we can ignore an event
- We want to find when we do not need to create, or can remove, a binding

Garbage

- When monitoring a garbage-collected language like Java there are two concerns with respect to garbage
  - The monitoring can cause *memory-leaks*
  - Some bindings may necessarily never lead to matches due to garbage values i.e. they are now redundant

- This was originally noted in early work on `tracematches`

- The typical solution is to use *weak references* to refer to monitored objects
- A weak reference in Java is ignored by garbage collection
- But we need to be careful...

Going Wrong with Weak References

- Consider the property *every file that is opened must be closed*
- What if a monitored file is in the open state and becomes garbage?
- Removing any reference to the file from the monitoring state would miss this violation

- It is important to detect the occurrence of garbage collection and treat the binding appropriately (see co-enable sets)
- Early work got this wrong (always read the most recent papers!)

Other Redundancy Issues

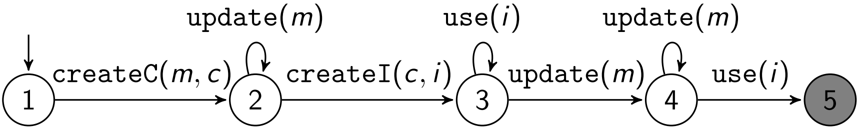
There are other notions of *redundancy* that can reduce the amount of work that you need to do.

- *creation events*: if every matching trace starts with a subset of events then start monitoring on these events only
- *enable sets*: for each event detect the set of other events that must occur first for that event to make a difference. We call such a the *enable set*. For efficiency reasons we can approximate events by the parameters they bind.
- *co-enable sets*: a symmetric notion for removing bindings. Detect the parameters needed to exist to reach a goal state. If they all become garbage then the binding can be removed.

Enable sets are a special instance of a more general notion of redundancy where *an event is considered redundant if ignoring it always gives the same verdict*. Easy to compute but not yet clear how to apply this notion *efficiently* in general.

An Example of (co)Enable Sets

- creation event: without a `createC` we don't need to record anything
- enable set: unless m and c are bound, we can ignore i
- coEnable set: if i is garbage collected then we cannot reach state 5



Optimising Parametric Trace Slicing

In this part we will consider:

- Extensions to the *expressiveness* of the theory
- Indexing techniques to improve *efficiency*
- Notions of *redundancy* that reduce the work required
- Other pragmatic issues.

Other Pragmatic Issues

- Monitoring multiple properties
  - What if we want to monitor many (similar) properties at the same time?
  - There exists work on sharing parts of the monitoring (and results on what not to share)
- Signal and Continue Monitoring
  - Note we often talk about success and failure, but many systems talk about *matches*
  - Slicing gives a nice signal-and-continue approach where sets of parameters can fail in separation
- Explaining Failures
  - If we get a violation how do we report it, what information can we give?
  - Tracking the code points that generated events is expensive
  - Signal-and-continue is a coarse-grained notion of multiple failure reporting

Summary

- We can have a more expressive slicing-based language than JavaMOP
- Indexing is important. The most prominent approach is value-based
- Reducing expressiveness can lead to more efficient indexing
- Removing redundancies is important. Dealing with garbage is very important for online monitoring
- Ongoing research: comparing slicing to other languages
  - Can we automatically translate between them?
  - Can we transfer algorithm optimisations i.e. indexing and notions of redundancy?

Static Partial Evaluation of Monitors

In this part we will

- Motivate the use of static analysis through some examples
- Quickly revisit what *pointer analysis* is
- Outline the CLARA architecture
- Describe four static whole-program analyses

Static Partial Evaluation of Monitors

In this part we will

- Motivate the use of static analysis through some examples
- Quickly revisit what *pointer analysis* is
- Outline the CLARA architecture
- Describe four static whole-program analyses

Motivating Static Analysis

Q Does the following violate the UnsafeMapIterator property?

A

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  System.out.println("There are "+map.keySet().size()+
                    " unique keys");
}
```

Motivating Static Analysis

Q Does the following violate the UnsafeMapIterator property?

A No. There are no iterators created.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  System.out.println("There are "+map.keySet().size()+
                   " unique keys");
}
```

Motivating Static Analysis

Q Does the following violate the UnsafeMapIterator property?

A

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = Arrays.asList(args).iterator();
  while(iter.hasNext()){
      String arg = iter.next();
      if(map.containsKey(Integer.parseInt(arg)) &&
         map.containsValue(arg)){
          System.out.println(arg+" is a key and value");
      }
  }
}
```

Motivating Static Analysis

Q Does the following violate the UnsafeMapIterator property?

A No. No one slice contains all necessary events.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = Arrays.asList(args).iterator();
  while(iter.hasNext()){
      String arg = iter.next();
      if(map.containsKey(Integer.parseInt(arg)) &&
         map.containsValue(arg)){
          System.out.println(arg+" is a key and value");
      }
  }
}
```

Motivating Static Analysis

Q Does the following violate the UnsafeMapIterator property?

A

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.keySet().iterator();
  while(iter.hasNext()){
    Integer key = iter.next();
    System.out.println(key+" \t:\t"+map.get(key));
  }
}
```

Motivating Static Analysis

Q Does the following violate the UnsafeMapIterator property?

A No. There are no updates after iteration.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.keySet().iterator();
  while(iter.hasNext()){
    Integer key = iter.next();
    System.out.println(key+" \t:\t"+map.get(key));
  }
}
```

Motivating Static Analysis

Q Does the following violate the UnsafeMapIterator property?

A

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.valueSet().iterator();
  while(iter.hasNext()){
    Integer key = iter.next();
    if(map.containsKey(Integer.parseInt(arg))){
      map.remove(key);
    }
  }
}
```

Motivating Static Analysis

Q Does the following violate the UnsafeMapIterator property?

A Maybe. We cannot tell statically.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.valueSet().iterator();
  while(iter.hasNext()){
    Integer key = iter.next();
    if(map.containsKey(Integer.parseInt(arg))){
      map.remove(key);
    }
  }
}
```

Motivating Static Analysis

Q Does the following violate the UnsafeMapIterator property?
A

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.valueSet().iterator();
  map.insert(0,"empty");
  while(iter.hasNext()){
    Integer key = iter.next();
    if(map.containsKey(Integer.parseInt(arg))){
      map.remove(key);
    }
  }
}
```

Motivating Static Analysis

Q Does the following violate the UnsafeMapIterator property?

A Yes. This insertion must violate the property.

```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = map.valueSet().iterator();
  map.insert(0,"empty");
  while(iter.hasNext()){
    Integer key = iter.next();
    if(map.containsKey(Integer.parseInt(arg))){
      map.remove(key);
    }
  }
}
```

What do we want?

- To reduce the work required at runtime
- We already established this involves deciding which events to safely ignore
- In the context of AspectJ this removes removing *joinpoint*s

- Static partial evaluation is about statically deciding which events do not need to be recorded. In the limit we can decide if the property necessarily does or does not hold

A Quick Guide to Static Analysis

- *Intra vs Inter procedural*
  - Intraprocedural considers functions/methods in separation. Assumes other procedures exhibit all possible behaviours.
  - Interprocedural considers whole program (full call graph)
- *Flow sensitive/insensitive.*
  - sensitive: considers the order of statements
  - insensitive: considers the statements as unordered
- *Context sensitive/insensitive (interprocedural only).*
  - sensitive: keeps track of the context of procedure calls i.e. its calling parameters
  - insensitive; the set of all contexts is considered
- *Heap abstraction.*
  - For heap-based languages (e.g. Java) it is necessary to model dynamically allocated objects
  - This is typically done by allocation sites (`new`) where each site gives a *representative object*

Static Partial Evaluation of Monitors

In this part we will

- Motivate the use of static analysis through some examples
- Quickly revisit what *pointer analysis* is
- Outline the CLARA architecture
- Describe four static whole-program analyses

Pointer Analysis

- The aim of *points-to* analysis is to compute for a variable x the superset of representative objects that x (*may/must*) point to during execution
- There is a trade-off between *precision* and *efficiency*
  - Imprecision may overapproximate i.e. *may-points-to*
  - Imprecision may underapproximate i.e. *must-points-to*
- The imprecision can come from different sources (e.g. flow insensitivity, approximating recursion)

By Some Examples

flow-insensitive may_points_to$(x) = \{1, 2\}$

```
A x;
void f () {x = new A(); }// (1)
void g() {x = new A(); }// (2)
void main() {
    f ();
    g ();
    print (x);
}
```

By Some Examples

flow-insensitive may_points_to$(x) = \{1, 2\}$
flow-sensitive may_points_to$(x) = \{2\}$

```
A x;
void f () {x = new A(); }// (1)
void g() {x = new A(); }// (2)
void main() {
    f ();
    g ();
    print (x);
}
```

By Some Examples

intraprocedural analysis must assume the `iterator` calls may return the same values, it may return anything

```
x = c. iterator (); // (3)
y = c. iterator (); // (4)
...
```

By Some Examples

interprocedural context-insensitive

$$\text{may\_points\_to}(x) = \{5\}$$
$$\text{may\_points\_to}(y) = \{5\}$$

```
x = c. iterator (); // (3)
y = c. iterator (); // (4)
...
public Iterator iterator () {
    return new HashSetIterator (); // (5)
}
```

By Some Examples

interprocedural context-sensitive

$$\text{may\_points\_to}(x) = \{\langle 3, 5\rangle\}$$
$$\text{may\_points\_to}(y) = \{\langle 4, 5\rangle\}$$

```
x = c. iterator (); // (3)
y = c. iterator (); // (4)
...
public Iterator iterator () {
    return new HashSetIterator (); // (5)
}
```

By Some Examples

For may_points_to we merge object representatives at merge points. Note that the points-to set of a variable changes during execution, analysis is with respect to a statement.

```
i = c1. iterator (); // 1
j = i;
if (p)
    i = c2. iterator (); // 2
// 3 =  1, 2
j = i;
print (j );
```

Static Partial Evaluation of Monitors

In this part we will

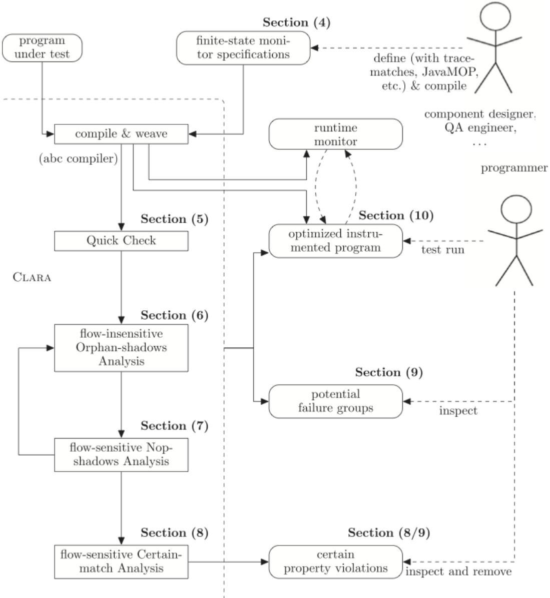
- Motivate the use of static analysis through some examples
- Quickly revisit what *pointer analysis* is
- Outline the CLARA architecture
- Describe four static whole-program analyses
- Give some further context as the above is relatively lightweight

CLARA

- A framework developed by Eric Bodden (with collaborators along the way) for his PhD thesis (2009)
- The main work to date on static partial evaluation of monitors
- Stands for ComiLe-time Approximation of Runtime Analyses

- The basic underlying ideas are:
  - Take monitors described using AspectJ aspects
  - Abstract the notion of finite-state monitors as *dependency state machines* and use to annotate aspects
  - Apply three staged static analyses to remove instrumentation points shown to be ineffectual
  - Apply a static analysis that detects certain violations

Architecture



Dependency State Machine

- CLARA assumes the monitor admits a finite state machine capturing dependencies between pointcuts
- It calls such machines *dependency state machines* (DSM)
- These machines should define the matching (bad) behaviours
- But they are just used for static analysis and do not include any *actions* to be taken on a match
- DSM are non-deterministic to support multiple matches i.e. every trace prefix leading to a final state should be matched (important when deciding what joinpoints to drop)

What are JoinPoints?

- A joinpoint is an instance of a pointcut p
- i.e. it is a statement s in the code where the pointcut matches
- A joinpoint-label label(s) is the DSM symbol defined by p

- A joinpoint associates some program variables with the pointcut parameters, these variables have points-to sets
- Let a joinpoint-binding $\beta(s)$ be a binding from pointcut parameters to sets of object representatives

- Two joinpoint- bindings are *compatible* if their points-to sets on the joint-domain overlap i.e.

$$\text{compatible}(\beta_1, \beta_2) \equiv \forall v \in (\underline{dom}(\beta_1) \cap \underline{dom}(\beta_2)) . \beta_1(v) \cap \beta_2(v) \neq \emptyset$$

Soundness Condition

- An analysis is *sound* if whenever it removes a join point the same matches are found
- Formally, we ask each analysis to define a predicate necessaryTransition(α, τ, i) that must be true whenever removing joinpoint α at the i-th position of τ would change the matching status of trace τ
- Such a predicate has been defined for the following analysis and proved to hold

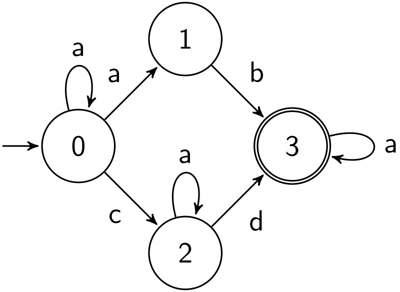
Static Partial Evaluation of Monitors

In this part we will

- Motivate the use of static analysis through some examples
- Quickly revisit what *pointer analysis* is
- Outline the CLARA architecture
- Describe four static whole-program analyses

Syntactic Quick Check

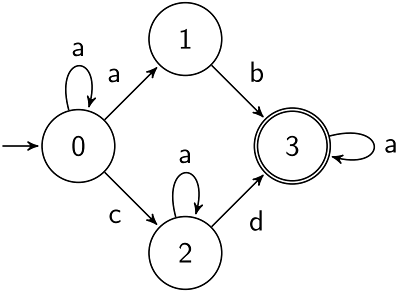
- The idea: check if the symbol needs monitoring at all



- Consider if. . .
- We only need to monitor. . .

Syntactic Quick Check

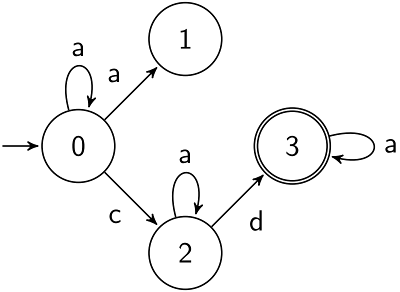
- The idea: check if the symbol needs monitoring at all



- Consider if. . . symbol b never occurs in the program
- We only need to monitor. . .

Syntactic Quick Check

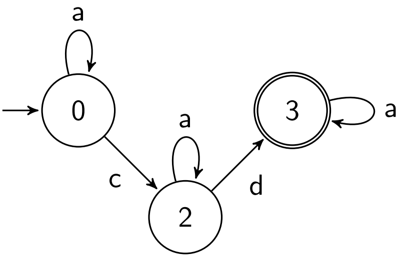
- The idea: check if the symbol needs monitoring at all



- Consider if. . . symbol b never occurs in the program
- We only need to monitor. . .

Syntactic Quick Check

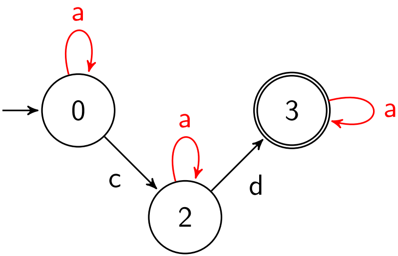
- The idea: check if the symbol needs monitoring at all



- Consider if. . . symbol b never occurs in the program
- We only need to monitor. . .

Syntactic Quick Check

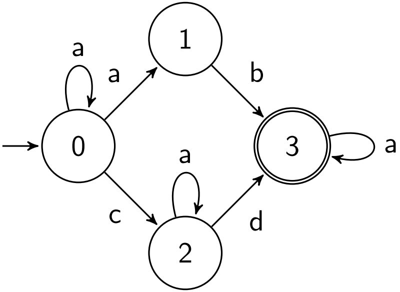
- The idea: check if the symbol needs monitoring at all



- Consider if. . . symbol b never occurs in the program
- We only need to monitor. . . c,d

Syntactic Quick Check

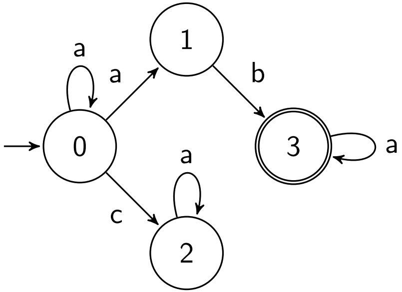
- The idea: check if the symbol needs monitoring at all



- Consider if. . . symbol d never occurs in the program
- We only need to monitor. . .

Syntactic Quick Check

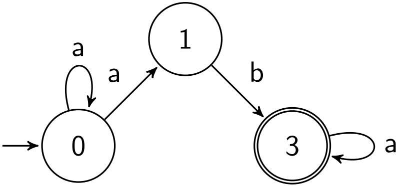
- The idea: check if the symbol needs monitoring at all



- Consider if. . . symbol d never occurs in the program
- We only need to monitor. . .

Syntactic Quick Check

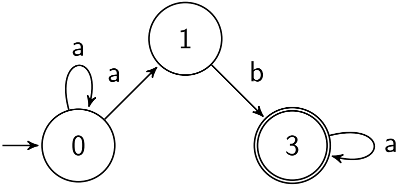
- The idea: check if the symbol needs monitoring at all



- Consider if. . . symbol d never occurs in the program
- We only need to monitor. . . a,b,c . . . why c? . . . consider acb

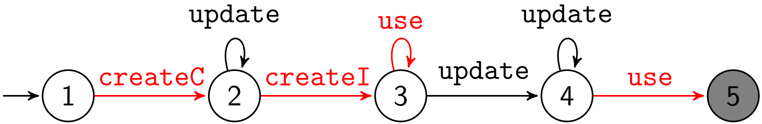
Syntactic Quick Check

- The idea: check if the symbol needs monitoring at all



- Consider if. . . symbol d never occurs in the program
- We only need to monitor. . .
- This is flow-insensitive (but interprocedural)

Motivating Static Analysis




```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  System.out.println("There are "+map.keySet().size()+
                  " unique keys");
}
```

Orphan-shadows Analysis

- The idea: perform the Quick Check 'per slice'
- Slices (e.g. bindings) are statically approximated using points-to set abstraction of joinpoints
- For each joinpoint s define the set of compatible symbols

$$\mathsf{compSyms}(s) \equiv \{\mathsf{label}(s') \mid \mathsf{compatible}(\beta(s), \beta(s'))\}$$

- A joinpoint s is necessary if

$$\mathsf{label}(s) \in \mathsf{QuickCheck}(\mathsf{compSyms}(s))$$

i.e. it is syntactically relevant when only considering possibly compatible slices

Orphan-shadows Analysis

- The idea: perform the Quick Check 'per slice'
- Slices (e.g. bindings) are statically approximated using points-to set abstraction of joinpoints
- For each joinpoint s define the set of compatible symbols

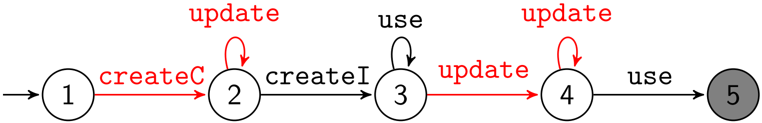
$$\text{compSyms}(s) \equiv \{\text{label}(s') \mid \text{compatible}(\beta(s), \beta(s'))\}$$

- A joinpoint s is necessary if

$$\text{label}(s) \in \text{QuickCheck}(\text{compSyms}(s))$$

  i.e. it is syntactically relevant when only considering possibly compatible slices
- CLARA uses interprocedural context-sensitive flow-insensitive points-to analysis

Motivating Static Analysis




```
public static void main(String args[]){
  Map<Integer,String> map = new HashMap<>();
  for(int i=0; i+1<args.length;i+=2){
      map.insert(Integer.parseInt(args[i]),args[i+1]);
  }
  Iterator iter = Arrays.asList(args).iterator();
  while(iter.hasNext()){
      String arg = iter.next();
      if(map.containsKey(Integer.parseInt(arg)) &&
         map.containsValue(arg)){
          System.out.println(arg+" is a key and value");
      }
  }
```

Nop-shadows Analysis

- The idea: compute, for each joinpoint, what state we could be in at that point, and which states could (hot) and could not (cold) lead to a match (final state) from that point
- We must keep a joinpoint if
  - It can transition from a hot to a cold state
  - It can transition from a cold to a hot state
- If we remove any such joinpoints we can get false positives and false negatives

A little more detail

- For a joinpoint s we define
  - futures(s) as sets of reachable states by backward analysis
  - sources(s) as reached states by forward analysis
  - target(q, s) as the state reached from q by s
- Then a joinpoint is a nop if

$$\forall q \in \text{sources}(s). q \equiv_s \text{target}(q, s) \wedge \text{target}(q, s) \notin F$$

where

$$q \equiv_s q' \quad \textit{iff} \quad \forall Q \in \text{futures}(s). q_1 \in Q \Leftrightarrow q_2 \in Q$$

- The analysis is intraprocedural but has some extra stuff to make things a little more precise

Certain-match Analysis

- The forward analysis computes the set of states reached by a statement
- If a statement necessarily reaches only final states then we have statically determined that there will certainly be a match

- Therefore this analysis can borrow this information from the previous analysis and find such certain matches for free

Summary

- Static partial evaluation can optimise slicing-based approaches by reducing the number of monitored events
- Question: can we apply the same ideas to more expressive notions of slicing (QEA)
- Question: can we apply the same ideas to different formalisms (non-automata based)

Evaluating Runtime Monitoring Tools

In this part we will cover

- The question of how we should evaluate RV tools
- Typical approaches to evaluation in the literature
- The Runtime Verification Competition
- Issues to consider when benchmarking

Evaluating Runtime Monitoring Tools

In this part we will cover

- The question of how we should evaluate RV tools
- Typical approaches to evaluation in the literature
- The Runtime Verification Competition
- Issues to consider when benchmarking

Evaluation

- Firstly we need to define what kind of tools we're dealing with
  - As you will have heard, RV is a broad term
  - Here we mainly consider *trace-checking* but some of the questions apply to RV (and other areas) more broadly

- Some questions
  - What aspects of the monitoring should we *measure*?
  - What kind of workloads do we want, how do we know if they are representative?
  - How do we compare with other techniques?
  - How does the monitoring setup affect how we evaluate? e.g. offline vs online, matching vs violations
  - Does reproducibility matter? (think concurrency)
  - What matters... e.g. overall overhead vs responsiveness?

Evaluation

- Firstly we need to define what kind of tools we're dealing with
  - As you will have heard, RV is a broad term
  - Here we mainly consider *trace-checking* but some of the questions apply to RV (and other areas) more broadly

- Some questions   Discuss
  - What aspects of the monitoring should we *measure*?
  - What kind of workloads do we want, how do we know if they are representative?
  - How do we compare with other techniques?
  - How does the monitoring setup affect how we evaluate? e.g. offline vs online, matching vs violations
  - Does reproducibility matter? (think concurrency)
  - What matters... e.g. overall overhead vs responsiveness?

The Big Issue

- Almost every RV tool has its own specification language

The Big Issue

- Almost every RV tool has its own specification language
- Some research has tried to look at translations between languages but there has not been much appetite in the research community

The Big Issue

- Almost every RV tool has its own specification language
- Some research has tried to look at translations between languages but there has not been much appetite in the research community - Discuss why

The Big Issue

- Almost every RV tool has its own specification language
- Some research has tried to look at translations between languages but there has not been much appetite in the research community - Discuss why
- What issues do we think this brings, what solutions might there be?

The Big Issue

- Almost every RV tool has its own specification language
- Some research has tried to look at translations between languages but there has not been much appetite in the research community - Discuss why
- What issues do we think this brings, what solutions might there be? - Discuss

Offline Setting

- Checking a single log file
- Generally only interested in the level of resources required

- Measure: how much *time* and *memory* required
- Possibly per-event but usually just totals

- Standard trace file formats emerging, making it easier to compare tools (see competition)

- So relatively straightforward

Online Setting

- There will be an unmonitored program that uses its own resources, say it takes $\mathcal{T}$ seconds to run

- Measure: new resources needed, say it now takes $\mathcal{M}$ seconds
- Important metrics:
  - **Overhead**: the amount of extra time needed
    - Could be raw i.e. $\mathcal{O} = \mathcal{M} - \mathcal{T}$
    - Often given as a percentage i.e. $100 \times \frac{\mathcal{O}}{\mathcal{T}}$
  - **Throughput** i.e. events per second
    - Might change during monitoring
  - **Responsiveness**: amount of time to process each event
    - As well as mean should include max and standard deviation etc
    - Might break down per event-type

- We might be able to break down overhead by type:
  - Instrumentation
  - Monitor evaluation
  - Synchronisation (especially with concurrent programs)

What makes online monitoring harder?

- Is instrumentation part of monitoring?
- Are we evaluating the instrumentation or the monitoring algorithm?

- How stable is the underlying program or the monitoring algorithm (how many times do we need to run this)?
- Are we evaluating noise in the underlying runtime system or the monitoring program?

- We might also care about **Interference** i.e. how has the execution of the program changed due to monitoring (reordered threads, different GC behaviour, energy profile). How do we measure this?

False Positive Rate

- If the analysis is precise then incorrect results suggest *unsoundness*, this is very bad

- If the analysis is imprecise then we can measure its *accuracy* i.e. how often it gets the correct result

- Typically we want to break this down as
  - False positive: identified a match when it wasn't a match
  - False negative: missed a match
- Why is the second one hard to measure?

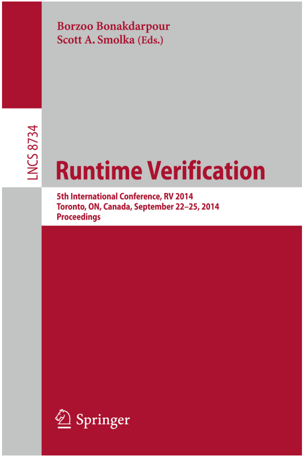
- We can also talk about whether identified bugs are really bugs. . . what is this measuring?

Evaluating Runtime Monitoring Tools

In this part we will cover

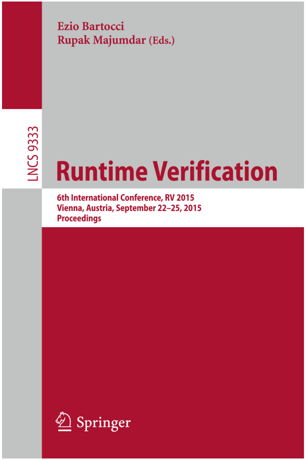
- The question of how we should evaluate RV tools
- Typical approaches to evaluation in the literature
- The Runtime Verification Competition
- Issues to consider when benchmarking

Benchmarks in the Literature



- Looking at proceedings of RV14 and RV15
- In 2014 (out of 27 papers)
  - 5 described monitoring algorithms
  - 7 described implementations
  - 17 had evaluation sections
  - 1 was a case study papers
  - 3 had data available online
- In 2015 (out of 21 papers)
  - 6 described monitoring algorithms
  - 7 described implementations
  - 11 had evaluation sections
  - 2 were case study papers
  - 3 had data available online

Benchmarks in the Literature



- Looking at proceedings of RV14 and RV15
- In 2014 (out of 27 papers)
  - 5 described monitoring algorithms
  - 7 described implementations
  - 17 had evaluation sections
  - 1 was a case study papers
  - 3 had data available online
- In 2015 (out of 21 papers)
  - 6 described monitoring algorithms
  - 7 described implementations
  - 11 had evaluation sections
  - 2 were case study papers
  - 3 had data available online

Benchmarks in the Literature

- (Correct me if I missed something, this is very broad)

- Of the above 28 evaluations no two papers used the same benchmarks
- No evaluation section made a comparison with another technique or tool (unless it was a previous version of the discussed one)
- Many (definitely not all) case studies were created for the evaluation

- This isn't very encouraging (I am not innocent)

DaCapo

- There are some 'standard' benchmarks frequently used
- One popular set is DaCapo, see `http://dacapobench.org/`

- Open source, real world applications with non-trivial memory loads
- Originally designed to evaluate JVMs and architectures

- Okay, from an RV perspective this has a very restrictive set of workloads and monitorable properties

Evaluating Runtime Monitoring Tools

In this part we will cover

- The question of how we should evaluate RV tools
- Typical approaches to evaluation in the literature
- The Runtime Verification Competition
- Issues to consider when benchmarking

The RV Competition

- Started in 2014 and ran in 2015 and 2016
- Goals: to improve benchmarking and tool comparison, and to drive research

- Has evaluated 14 different tools
- Has used over 70 different benchmarks (some similar)

- Measured time and memory utilisation
- Split into online C, online Java and Offline

- We briefly discuss the tracks

The C Track

- The most problematic track. This track didn't run in 2016 due to lack of interest

- Attracted interest from the static community, but their notion of property was very different
- Traditional RV concentrates on explicit temporal properties (i.e. in LTL) whereas the static community (who joined in) focuses on implicit properties (memory safety) and assertions

- Suffered from a lack of well established tools for monitoring C programs
- There may be a relatively high barrier to entry due to a lack of well-used instrumentation methods within the community

The Java Track

- Only a few players generally monitoring well-known/standard properties
- Some benchmarks just replay trace files (I'm guilty of this)
- This an lead to artificially high overhead (all the work is monitoring)
- Massive variations in results (a few seconds vs a few hours) mainly attributed to improper handling of garbage
- One success: the MUFIN tool was developed with the purpose of winning this track, and they did. So the competition led to knew research.

The Offline Track

- Surprisingly (maybe) the most popular track
- Probably because of low barrier to entry (just need to parse traces)

- The competition introduced various trace formats, which have evolved
- The most popular format was CSV, but there were some issues with this for more structured data

- Almost completely automated evaluation (the other tracks required a bit of manual work to set up)

What can we do better?

- The competition should serve the research community but also act as an incentive to explore new areas
- What format do you think it should take?
- What should we be measuring? Is time really that important?
- How do we encourage teams to take part?
- How do we deal with
  - No common specification language (are submitted monitors equivalent?)
  - No common instrumentation techniques (what are we measuring?)

Evaluating Runtime Monitoring Tools

In this part we will cover

- The question of how we should evaluate RV tools
- Typical approaches to evaluation in the literature
- The Runtime Verification Competition
- Issues to consider when benchmarking

Issues to Consider

- Are you measuring what you care about?
  - Does overhead matter in your scenario?
  - Does the evaluation actually measure whether you solved the targeted problem?
- Are your results significant?
  - How do they compare to other techniques?
  - Are you using realistic workloads?
  - Are your benchmarks big enough? (the JVM startup effect)
- Are your results reproducible?
  - Are the benchmarks downloadable?
  - Do you report on the whole setup (e.g. memory limits)
  - Are the results stable (error bars)