Some logistics

• Homework-0 is assigned, due 9/15 11:59pm AoE
  • Goal: getting your hands wet with RV

• Reading for next class is assigned
  • Goal prepare for more foundational topics in RV

• Thanks for submitting reading 1; we’ll discuss some of your responses in class
Concepts discussed in this class

• Runtime verification checks *traces* of system *events* against *properties*

• But what do the terms in *blue* mean?

• These terms occur a lot in the RV literature
Let’s discuss...

• What is an event?
  observable change of state
  abstract set of things we care about

• What is a trace?
  series of ordered events
  can be p.o. purpose is to describe execs

• What is a property?
  set of traces that meet some condition
  membership
What is an Event?

• A mathematical (formal languages) view
  • An event as a symbol $e$ in an alphabet $\Sigma$, where $\Sigma$ is a finite set of such symbols

• A logical view
  • An event as an atomic predicate in a logical formula

• A practical view
  • An event as a state in system/software execution
When/how you’ll see these views

• View of events as symbols is common when defining concepts or proving theorems in RV

• View of events as atomic predicates is often used when specifying properties

• View of events as execution steps is required when defining what to observe during system execution
Example: CSC spec from last class

```java
public static <T> Collection<T> synchronizedCollection(Collection<T> c)

It is imperative that the user manually synchronize on the returned collection when iterating over it:

Collection c = Collections.synchronizedCollection(myCollection);
...

synchronized (c) {
    Iterator i = c.iterator(); // Must be in the synchronized block
    while (i.hasNext())
        foo(i.next());
}

Failure to follow this advice may result in non-deterministic behavior.
```

• What events (execution states) do we care about?
Example: events in the CSC spec
http://www.kframework.org/tool/run/javamop

1. Click on spec
What (view of) events are in CSC?

- Logical view
- Practical view
Events as execution states

- Examples: method calls, field/variable access, lock acquisition/release
- One often must define the conditions under which to observe the execution step

```java
21: event syncCreateIter after(Object c) returning(Iterator iter):
22:     call(* Collection+.iterator())
23:     && target(c) && if(Thread.holdsLock(c)){}
24:```

- Events can carry data, or they can be parametric
What view of events is this? (1)

- A property is a logical formula over a set of events

<sup>1</sup>Legunsen et al., Techniques for Evolution-Aware Runtime Verification, ICST 2019
What view of events is this? (2)

An RV tool instruments the program based on the properties so that executing the instrumented program generates events and creates monitors that listen to events and check properties?¹

¹Legunsen et al., Techniques for Evolution-Aware Runtime Verification, ICST 2019
What view of events is this? (3)

• A bad prefix is a finite sequence of events which cannot be the prefix of any accepting trace.²

²d’Amorim et al., Efficient Monitoring of ω-Languages, CAV 2005
Takeaway message on events

• Events are fundamental in RV theory and practice

• But RV literature will often mix the different views of events

• So, when you read papers on RV, be careful to distinguish these views
Any questions about events?
What is a trace?

There are many notions/views of traces in RV, e.g.,

What Is a Trace? A Runtime Verification Perspective

Giles Reger$^{1(16)}$ and Klaus Havelund$^{2}$

$^1$ University of Manchester, Manchester, UK
giles.reger@manchester.ac.uk

$^2$ Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA

What is a trace? Some views..

• A trace is a sequence of events
  • In practice: sequences are finite
  • In theory: we reason about infinite sequences

• If events are symbols in an alphabet $\Sigma$, traces are strings (or words) in $\Sigma^*$
  • We can talk about finite prefixes or suffixes of traces
What is a trace? (A definition)

Let $\Sigma$ be a set of events. A $\Sigma$-trace (or simply a trace when $\Sigma$ is understood or not important) is any finite sequence of events in $\Sigma$, that is, an element in $\Sigma^*$. If event $e \in \Sigma$ appears in trace $w \in \Sigma^*$ then we write $e \in w$.\(^3\)

\(^3\)Rosu and Chen, Semantics and Algorithms for Parametric Monitoring, LMCS 2012
Example 1: events and traces

Consider a resource (e.g., a synchronization object) that can be acquired and released during the lifetime of a procedure (i.e., between when the procedure begins and when it ends).

1. What events do we care about?

   acquire release begin end

2. What is an example trace over events in 1?

   acquire & begin release end
“Good” and “bad” traces

• In example 1 context, are these good or bad traces:
  • begin acquire release end
  • begin acquire acquire release end
  • begin acquire acquire release release end

• Properties formalize notion of “good” or “bad” traces

• Terminology: traces validate or violate a property depending on how the property is expressed
What is a property?

• A property is a set of traces
  • may include “good” traces and exclude “bad” traces
  • or, it may exclude “good” traces and include “bad” traces

• Alternately, a property is a language of acceptable or unacceptable traces (a subset of $\Sigma^*$).

• In practice, can you think of why set/language inclusion/exclusion may be insufficient for RV?
Are these definitions sufficient?

• If “good” properties in example 1 are those in which an acquired resource is released before the procedure ends. Are these “good” or “bad” traces?
  • begin acquire release acquire end
  • begin acquire acquire

• Partial traces may be in “don’t know” category
  • future events may lead to including/excluding the trace

• We need to build on the idea of partitioning traces into categories
Properties: another definition

An \( \Sigma \)-property \( P \) (or simply a \textit{property}) is a function \( P : \Sigma^* \rightarrow C \) partitioning the set of traces into (verdict) categories \( C \).

- This definition is more general (in a sense)
  - \( C \) can be any set, e.g., \{validating, violating, don’t-know\}
  - \( C \) is chosen depending on the specification language and the property being specified
Properties partition sets of traces (1)

• Let regular expressions (RE) be the spec language and choose $C = \{\text{match, fail, dont-know}\}$

• Then for any RE, $E$, its property $P_E$ can be defined as
  • $P_E(w) = \text{match}$ iff $w$ is in the language of $E$
  • $P_E(w) = \text{fail}$ iff $\not\exists w' \in \Sigma^* \text{ s.t. } \text{ww'}$ is in the language of $E$
  • $P_E(w) = \text{dont-know}$ otherwise

• This is the semantics of monitoring RE in JavaMOP
Properties partition sets of traces (2)

• Let regular expressions (RE) be the spec language and choose \( C = \{ \text{match}, \text{dont-care} \} \)

• Then for any RE, \( E \), its property \( P_E \) can be defined as
  • \( P_E(w) = \text{match} \) iff \( w \) is in the language of \( E \)
  • \( P_E(w) = \text{dont-care} \) otherwise

• This is the semantics of monitoring RE in tracematches\(^4\)

\(^4\)Allan et al., Adding Trace Matching with Free Variables to AspectJ, OOPSLA 2005
Demo: generating CSC traces
http://www.kframework.org/tool/run/javamop

1. Click on spec
2. Modify spec
3. Click on code
4. Run with RV
What we saw during the demo

• CSC specifies “bad” traces as a regular expression:
  • (sync asyncCreateIter) | (sync syncCreateIter accessIter)

• Matching trace from SafeSyncCollection_1.java:
  • sync asyncCreateIter accessIter accessIter accessIter accessIter

• Matching trace from SafeSyncCollection_2.java:
  • sync syncCreateIter accessIter accessIter accessIter accessIter accessIter
Properties: other things to know

• Can all interesting system behavior be defined as “sets of traces”?  
  • No. Hyperproperties\(^5\) are “sets of sets of traces”.

• Properties are sometimes called “trace properties”  
  • In contrast with “state properties”, which are defined in terms of program values at a point in an execution  
  • xUnit Assertions are examples of “state properties”

\(^5\)Clarkson and Schneider, Hyperproperties, CSF 2008
Recall: events can be parametric

- Events in real programs occur on different “objects”
  
  ```
  Parameters
  13: SafeSyncCollection(Object c, Iterator iter) {
  14: }
  ```

- RV tools must be able to handle parametricity to correctly partition traces at runtime
  - Let’s look at an example
Acquire/release revisited

• Property: procedures must release acquired resources

• Spec: \( (\text{begin}(\epsilon | (\text{acquire}(\text{acquire}|\text{release})^*\text{release}))\text{end})^* \)
  
  Multiple “acquire” or “release” have the effect of acquiring or releasing the resource exactly once

• Categorize as a match, fail, or don’t-know (JavaMOP):

begin acquire acquire acquire release end begin acquire release end
Acquire/release revisited

• Same trace, but two different resources ($r_1$ and $r_2$):

begin{} acquire$r_1$ acquire$r_2$ acquire$r_1$ release$r_1$ end{} 
begin{} acquire$r_2$ release$r_2$ end{}

• Categorize this parametric trace (JavaMOP)
  • Your answer: fail
  • Reason: $r_2$ in first invocation of procedure is not released
Monitoring a parametric trace (1)

- Intuition: split into two trace slices, one per resource

begin\(\langle\) acquire\(\langle r_1 \rangle\) acquire\(\langle r_2 \rangle\) acquire\(\langle r_1 \rangle\) release\(\langle r_1 \rangle\) end\(\rangle\) begin\(\langle\) acquire\(\langle r_2 \rangle\) release\(\langle r_2 \rangle\) end\(\rangle\)

\[\downarrow\]

begin\(\langle\) acquire\(\langle r_1 \rangle\) acquire\(\langle r_1 \rangle\) release\(\langle r_1 \rangle\) end\(\rangle\) begin\(\langle\) end\(\rangle\) &

begin\(\langle\) acquire\(\langle r_2 \rangle\) end\(\rangle\) begin\(\langle\) acquire\(\langle r_2 \rangle\) release\(\langle r_2 \rangle\) end\(\rangle\)
Monitoring a parametric trace (2)

• Then, check the trace slices non-parametrically:

begin acquire acquire release end begin end

begin acquire end begin acquire release end
Parametric trace slicing

• Essential for monitoring real software

• Future discussion: definitions and algorithms for efficient trace slicing

• Defining parametric trace slicing and parametric monitoring needs definitions of
  • parametric events
  • parametric traces
  • parametric properties
Let $X$ be a set of **parameters** and let $V$ be a set of corresponding **parameter values**. If $\Sigma$ is a set of events, then let $\Sigma\langle X \rangle$ denote the set of corresponding **parametric events** $e\langle \theta \rangle$, where $e$ is an event in $\Sigma$ and $\theta$ is a partial function in $[X \rightarrow V]$. A **parametric trace** is a trace with events in $\Sigma\langle X \rangle$, that is, a string in $\Sigma\langle X \rangle^*$. 

- Revisit these definitions in the class on trace slicing
- You now have an intuition for when you see these terms in RV papers
Parametric properties: examples

• Releasing acquired resources ✓
• Authenticate before use
• Safe iterators
• Correct locking
Example: authenticate before use

- Property: keys must be authenticated before use
- LTL spec: $\forall k. (\square (use \rightarrow \diamond authenticate))$
- Parametric trace:
  
  authenticate\text{\langle}k_1\text{\rangle} authenticate\text{\langle}k_3\text{\rangle} use\text{\langle}k_3\text{\rangle} use\text{\langle}k_2\text{\rangle} authenticate\text{\langle}k_2\text{\rangle} use\text{\langle}k_1\text{\rangle} use\text{\langle}k_2\text{\rangle} use\text{\langle}k_3\text{\rangle} 

- $k_1$ trace slice:
- $k_2$ trace slice:
- $k_3$ trace slice:
Example: safe iterators

- **Property:** when an iterator is created for a collection, do not modify the collection while its elements are traversed using the iterator

- **Events:** create\(\langle c, i \rangle\) creates iterator \(i\) from collection \(c\), update\(\langle c \rangle\) modifies \(c\), and next\(\langle i \rangle\) traverses \(c\)'s elements using \(i\)

- **RE Spec:** \(\forall c, i.\ create\ next\ *\ update\ +\ next\)

- **Parametric trace:**
  create\(\langle c_1, i_1 \rangle\) next\(\langle i_1 \rangle\) create\(\langle c_1, i_2 \rangle\) update\(\langle c_1 \rangle\) next \(\langle i_1 \rangle\)
Example: safe iterators (your turn)

- **RE Spec:** $\forall c, \text{create} \ next * \text{update} + \text{next}$
- **Parametric trace:**
  create$\langle c_1, i_1 \rangle$ next$\langle i_1 \rangle$ create$\langle c_1, i_2 \rangle$ update$\langle c_1 \rangle$ next $\langle i_1 \rangle$
- **Questions:**
  - Is there a trace slice that violates the spec?
  - If “yes”, which pair(s) of parameters are in the slice?
What we discussed today

• What is an event?

• What is a trace?

• What is a property?

• What are parametric events, traces, and properties?

• Intro to parametric trace slicing (to be continued...)
Any questions about events, traces, and parameters?