Julien Brunel, David Chemouil, Alcino Cunha, Eunsuk Kang, Nuno Macedo

FORMAL SOFTWARE DESIGN WITH ALLOY AND ELECTRUM

TEMPORAL LOGIC

Universidade do Minho & INESC TEC

ONERA DTIS & Université fédérale de Toulouse

Carnegie Mellon University

3rd World Congress on Formal Methods, Porto, Portugal, October 2019

TEMPORAL LOGIC

LINEAR TEMPORAL LOGIC

- Electrum includes temporal connectives from Linear Temporal Logic (LTL).
- Formulas are interpreted over infinite sequences of states (traces).

FUTURE OPERATORS

Electrum	Meaning
`always` p	p is always true from now on
`eventually` p	p will eventually be true
`after` p	p will be true in the next state
p **`until`** q	q will eventually be and p is true until then
p **`releases`** q	q can only be false after p is true

PAST OPERATORS

Electrum	Meaning
historically p	p was always true
once p	p was once true
before p	p was true in the previous state
p **since** q	q was once true and p has been true afterwards
p **triggered** q	if p was once true, then q has been true onwards

SEMANTICS BY EXAMPLE

eventually B

SEMANTICS BY EXAMPLE

not eventually B

SEMANTICS BY EXAMPLE

always A

SEMANTICS BY EXAMPLE

not always A

SEMANTICS BY EXAMPLE

before B

SEMANTICS BY EXAMPLE

not before B

SEMANTICS BY EXAMPLE

once B

SEMANTICS BY EXAMPLE

once A

SEMANTICS BY EXAMPLE


```
always (B implies eventually A)
```

SEMANTICS BY EXAMPLE

not always (A **implies eventually** B)

SEMANTICS BY EXAMPLE

eventually always A

SEMANTICS BY EXAMPLE

not eventually always B

SEMANTICS BY EXAMPLE

always eventually A

SEMANTICS BY EXAMPLE

not always eventually B

SOME LTL VALID FORMULAS

```
always A iff not eventually not A

eventually always A implies always eventually A

A until B implies eventually B

A releases B iff (always not A) or B until (A and B)
```

TRASH

Design a trash such that:

- A deleted file can still be restored if the trash is not emptied

TRASH

```
assert restoreIsPossibleBeforeEmpty {
  --  a deleted file can still be restored if the trash is not emptied
  always (all f:File | delete[f] implies
            (empty releases restoreEnabled[f]))
}
```


DEMO

EXERCISE

https://github.com/haslab/Electrum2/wiki/Trash

(exercises 4)

INFINITE TRACES

WHY INFINITE TRACES ONLY?

Semantic issues with finite traces

- What formulas are true in the last state?
- Different possible semantics

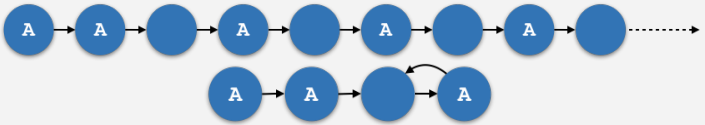
In Electrum: infinite traces only

- Caveat: no deadlock detection in general
- But a finite trace can be represented by an infinite one stuttering on the last state

FINITE REPRESENTATION WITH LASSOS

Lasso trace

- Some infinite traces can be represented by finite lasso traces



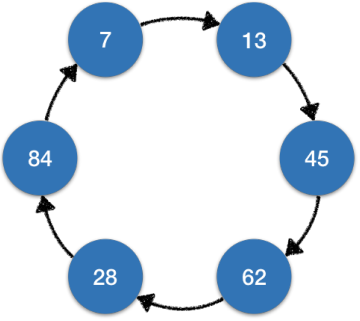
- Notice some infinite traces cannot

Small-Model Property for LTL

If an LTL formula is satisfiable, then it is satisfied by at least one lasso trace.

EXAMPLE

LEADER ELECTION IN A RING



Verify the correctness of the protocol:

- One leader will be elected

CONFIGURATION AND STATE

```
open util/ordering[Id]

sig Id {}

sig Node {
    id : one Id,
    succ : one Node,
    var inbox : set Id,
    var outbox : set Id
}

fact ring {
    all i : Id | lone id.i
    all n : Node | Node in n.^succ
}
```

ELECTION

```
fun elected : set Node {
    { n : Node | once n.id in n.inbox }
}
```

OPERATIONS

```
pred send [n : Node] { ... }

pred compute [n : Node] {
    some i : n.inbox {
        n.inbox' = n.inbox - i
        n.outbox' = n.outbox + (i - n.id.*(~next))
    }
    all m : Node - n | m.inbox' = m.inbox
    all m : Node - n | m.outbox' = m.outbox
}

pred skip { ... }
```

BEHAVIOR

```
fact init {
    no inbox
    outbox = id
}

fact transitions {
    always (skip or some n : Node | send[n] or compute[n])
}
```

EXERCISE

https://github.com/haslab/Electrum2/wiki/Leader-election

(exercise 1-3)

SAFETY VS LIVENESS

SAFETY PROPERTIES

- Something “bad” will never happen.
- A trace that violates a safety property has a “bad” prefix.
- A “bad” prefix is one s.t. every possible continuation violates the property.

```
always p
always (p implies once q)
always (p implies after always not p)
```

LEADER ELECTION

```
assert safety {
  always lone elected
}
```


DEMO

EXERCISE

https://github.com/haslab/Electrum2/wiki/Leader-election

(exercise 4)

LIVENESS PROPERTIES

- Something “good” will eventually happen.
- A property is a liveness property if any prefix can be extended to an infinite trace satisfying it.
- Much harder to check than safety properties.
  - Observing a prefix is not sufficient to detect a violation.

```
eventually p
always (p implies eventually q)
always eventually p
```

LEADER ELECTION

```
assert liveness {
  eventually some elected
}
```


DEMO

FAIRNESS

FAIRNESS ASSUMPTIONS

- Necessary for verifying most liveness properties.
- Exclude traces where an event becomes “continuously” enabled but never occurs
  - continuously = infinitely often (strong)
  - continuously = permanently (weak)

Strong fairness

```
(always eventually p) implies (always eventually q)
```

Weak fairness

```
(eventually always p) implies (always eventually q)
```

LEADER ELECTION SPECIFICATION FIXED

```
pred sendEnabled [n : Node] { some n.outbox }
pred computeEnabled [n : Node] { some n.inbox }
pred fairness {
  all n : Node {
    (eventually always sendEnabled[n]) implies
      (always eventually send[n])
    (eventually always computeEnabled[n]) implies
      (always eventually compute[n])
  }
}
assert liveness {
  fairness implies eventually some elected
}
```

CONCURRENT EXECUTIONS

INTERLEAVED VS. CONCURRENT EXECUTIONS

- Previous models force interleaved execution of operations
  - the frame condition in each event prevents other events from occurring
- Concurrent executions can also be specified in Electrum
  - frame conditions must be relaxed, but still prevent arbitrary changes
  - counter-examples can be shorter (more efficient verification) but harder to understand
  - some properties that are true in an interleaved model may no longer be true

OPERATIONS REDEFINED

```
pred skip { ... }
pred send [n : Node] { ... }
pred compute [n : Node] {
    some i : n.inbox {
        n.inbox' = n.inbox - i
        n.outbox' = n.outbox + (i - n.id.*(~next))
    }
}
```

FRAME CONDITIONS

```
fact frame {
    always all n : Node {
        n.inbox' != n.inbox implies (compute[n] or send[succ.n])
        n.outbox' != n.outbox implies (compute[n] or send[n])
    }
}
```


DEMO