Alcino Cunha

SPECIFICATION AND MODELING

ELECTRUM OVERVIEW

Universidade do Minho & INESC TEC

2019/20

DISTRIBUTED ATOMIC TRANSACTION PROTOCOL

- Several distributed *workers* performing a joint task
- If all succeed then all can *commit* the results
- If some aborts then all must *abort*
- After completing the task a worked is said to be *prepared* (to commit)

Design a protocol such that:

- It is never the case that we have both committed and aborted workers
- Once committed a worker stays committed (same for aborted)
- Once one worker commits all will eventually commit

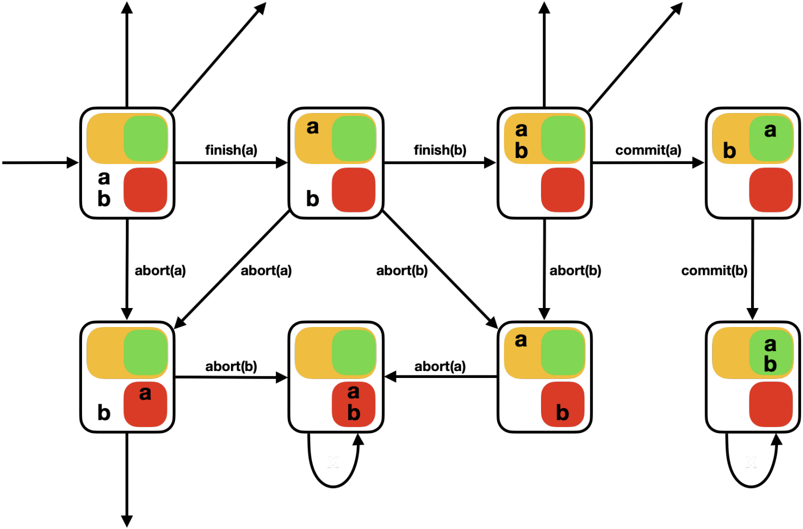
OUR TASKS FOR TODAY

- Design (a very abstract version of) the protocol
- Validate the design using simulation
- Verify that the design satisfies the expected properties

MODELLING WITH TRANSITION SYSTEMS

- A reactive or distributed system design can be modelled with a *transition system*
- *States* capture the global status of the system entities and environment
- *Transitions* originate from events performed by entities or the environment
- All states are assumed to always have at least one outgoing transition

THE PROTOCOL TRANSITION SYSTEM



STATE MODELLING

- A *state* is an assignment of values to variables
- In abstract design, use standard mathematical structures for variables

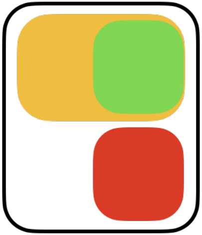
Alloy

- State is modelled just with sets and relations
- Inhabited by (tuples of) uninterpreted *atoms*
- Sets are declared with the `sig` keyword

Electrum

- Mutable sets and relations are declared with the `var` keyword

THE PROTOCOL STATE



```
sig Worker {}
var sig Prepared in Worker {}
var sig Committed in Prepared {}
var sig Aborted in Worker {}
```

EXPLICIT BEHAVIOUR MODELLING

- A transition system behaviour can be modelled explicitly:
  - Define which are the initial states
  - Define, for each event, how the next state(s) can be obtained from the current one

SMV

The transition system behaviour is explicitly modelled with a DSL

SMV can detect *deadlocks*, states without outgoing transitions

IMPLICIT BEHAVIOUR MODELLING

- The behaviour of a transition system can be abstracted by its set of *infinite traces*
  - This is known as a *linear model of time*
- This set of traces can be modelled implicitly:
  - By a property that "recognises" the valid traces among all possibles sequences of states
  - This property can be specified with a *linear temporal logic*
  - Combined with *first order logic* to specify assertions about states

Electrum + TLA

The transition system behaviour is implicitly modelled with a first-order temporal logic specification

Electrum

The specification is enclosed in a `fact`
The (infinite) traces satisfying this specification are also known as *instances*

FIRST ORDER LOGIC

Alloy	Math
not	$\neg$
and	$\wedge$
or	$\vee$
implies	$\longrightarrow$
all x : e | p	$\forall x \cdot x \in e \longrightarrow p$
some x : e | p	$\exists x \cdot x \in e \wedge p$

SET OPERATORS

Alloy	Math
`in`	$\subseteq, \in$
`+`	$\cup$
`&`	$\cap$
`-`	$\setminus$
`no e`	$e = \emptyset$
`some e`	$e \neq \emptyset$

LINEAR TEMPORAL LOGIC

Electrum	Meaning
`always` `p`	`p` is always true from now on
`after` `p`	`p` is true in the next state
...	...
`e'`	the value of `e` in the next state

AN ELECTRUM PATTERN FOR BEHAVIOUR SPECIFICATION

```
fact init { ... }
fact transitions { always (event1 or event2 or ...) }
```

- The specification of every event typically involves:
  - *Guard* - a state formula that checks if the event can occur
  - *Effect* - a formula with primes specifying how some state variables change
  - *Frame* - a formula with primes stating what does *not* change

THE PROTOCOL BEHAVIOUR

```
fact init { no Prepared and no Aborted }

fact transitions {
  always (
    // finish
    (some w: Worker | w not in Prepared and            -- guard
                      Prepared'  = Prepared + w and    -- effect
                      Committed' = Committed and       -- frame
                      Aborted'   = Aborted) or
    // commit
    ... or
    // abort
    ...
  )
}
```

THE PROTOCOL BEHAVIOUR REFACTORED WITH PREDICATES

```
pred finish[w : Worker] {
    w not in Prepared
    Prepared'  = Prepared + w
    Committed' = Committed
    Aborted'   = Aborted
}

pred abort[w : Worker] {
    w not in Aborted
    w in Prepared implies some Aborted
    Prepared'  = Prepared - w
    Committed' = Committed
    Aborted'   = Aborted + w
}
```

THE PROTOCOL BEHAVIOUR REFACTORED WITH PREDICATES

```
pred commit[w : Worker] {
    w in Prepared-Committed
    no Aborted
    Prepared'  = Prepared
    Committed' = Committed + w
    Aborted'   = Aborted
}

fact transitions {
  always (
    some w : Worker | finish[w] or commit[w] or abort[w]
  )
}
```

SIMULATION

- Models include analysis commands
- A **`run`** command asks for an instance (checking the consistency of the facts)
- Further instances can be obtained by an interactive exploration mode akin to simulation
- All commands have a scope that bounds the size of the signatures
- The default is 3, but can be changed with the **`for`** keyword


DEMO

THE FIXED PROTOCOL BEHAVIOUR

```
pred finish[w : Worker] {
    w not in Prepared
    w not in Aborted
    Prepared'  = Prepared + w
    Committed' = Committed
    Aborted'  = Aborted
}
```

THE FIXED PROTOCOL BEHAVIOUR

```
pred nop {
    Prepared'  = Prepared
    Committed' = Committed
    Aborted'   = Aborted
}

fact transitions {
  always (
    nop or some w : Worker | finish[w] or commit[w] or abort[w]
  )
}
```

ASSERTIONS

- In Electrum, the same first order temporal logic is used for
  - modelling the transition system
  - specifying the expected properties (*assertions*)
- The latter can be enclosed in named `assert` paragraphs

THE PROTOCOL ASSERTIONS

```
assert Consistency {
    -- It is never the case that we have both committed
    -- and aborted workers
    always (no Committed or no Aborted)
}

assert Stability {
    -- Once committed a worker stays committed (same for aborted)
    all w : Worker {
        always (w in Committed implies always w in Committed)
        always (w in Aborted implies always w in Aborted)
    }
}
```

VERIFICATION

- **check** commands are used to verify assertions
- The verification is fully automatic, but limited to the specified scope
- The set of counter-examples can be explored likewise instances


DEMO

THE FIXED PROTOCOL BEHAVIOUR

```
pred commit[w : Worker] {
    Worker in Prepared
    Prepared'  = Prepared
    Committed' = Committed + w
    Aborted'   = Aborted
}
```