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

FORMAL SOFTWARE DESIGN WITH ALLOY AND ELECTRUM

OVERVIEW

Universidade do Minho & INESC TEC

ONERA DTIS & Université fédérale de Toulouse

Carnegie Mellon University

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

INTRODUCTION

FORMAL SOFTWARE DESIGN

- A software design is a high-level abstraction of a desired system
- A programming language is not adequate for *analyzing* and *exploring* designs
- The language of mathematics, logic, is a much better alternative

> **Leslie Lamport**
>
> *"If you're not writing a program, don't use a programming language"*

TYPICAL ANALYSES

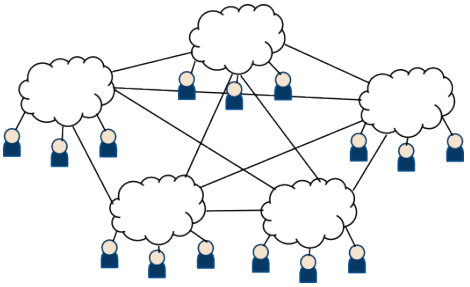
- Elicit requirements or structural constraints over a design
- Simulate a design to understand it
- Check consistency of requirements
- Check that some structural, invariant or temporal properties hold

Alloy and Electrum

- Alloy is great for the structural aspects, but has limitations for behavioral design
- Electrum extends Alloy to overcome some of these limitations

EXAMPLES

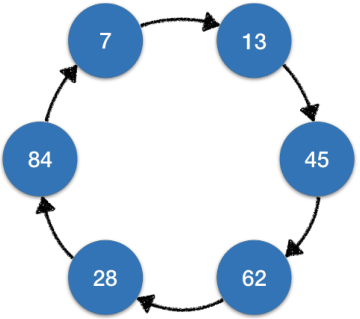
DISTRIBUTED SOCIAL NETWORK



Explore design alternatives

- What distributed data model and replication strategy to use?

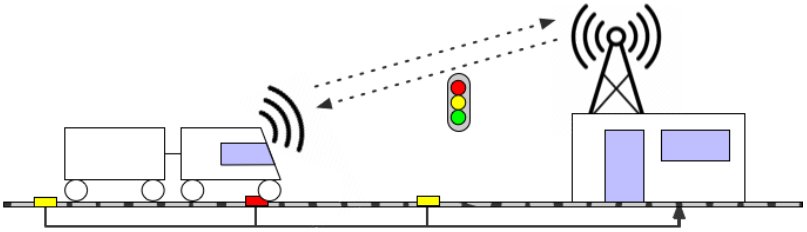
LEADER ELECTION IN A RING



Verify the correctness of the protocol:

- One leader will be elected

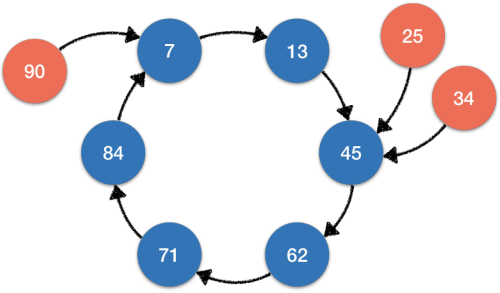
HYBRID ERTMS/ETCS LEVEL 3



Verify the design of a railway traffic management system:

- Assigned movement authorities are safe

CHORD DISTRIBUTED HASH-TABLE



Explore variants of the protocol and verify correctness:

- If joins and failures cease, the network will eventually become a ring

AN OVERVIEW WITH THE TRASH EXAMPLE

TRASH

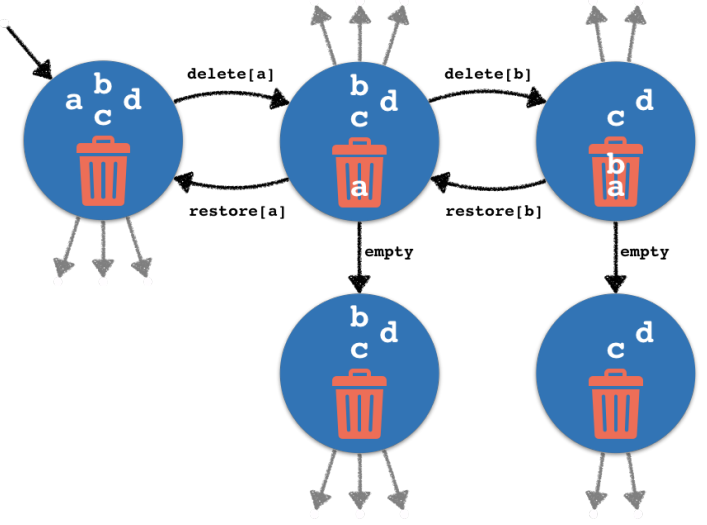
Design a trash such that:

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

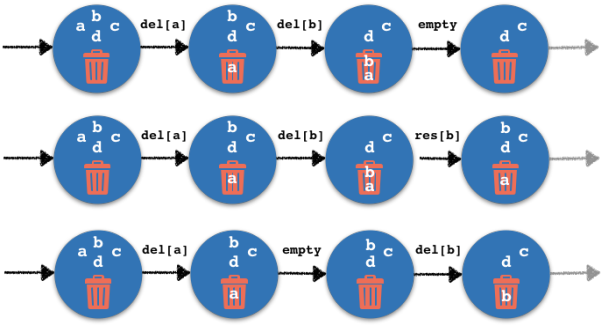
TASKS

- Design the structure and behavior (operations)
- Validate this design by simulation
- Elicit and verify expected properties

TRANSITION SYSTEMS



FROM TRANSITION SYSTEMS TO SETS OF TRACES



Electrum

- *Linear* model of time: sets of *infinite traces* (a.k.a *instances*)
- *Intentional* specification: formulas

STATE

- A *state* is an assignment of values to variables
- In abstract design, it is useful to rely on standard mathematical structures

Alloy and Electrum

- Values are sets and relations
- Inhabited by (tuples of) uninterpreted *atoms*

TRASH STATE

```
var sig File {}
var sig Trash in File {}
```

A PATTERN FOR BEHAVIOR FORMULAS

```
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

TRASH BEHAVIOR

```
fact init { no Trash }

fact transitions {
  always (
    // delete file
    (some f: File | f not in Trash and
                    Trash' = Trash + f and
                    File' = File) or
    // restore file
    ... or
    // empty trash
    ...
  )
}
```

TRASH BEHAVIOR REFACTORED

```
pred delete[f : File] {
  f not in Trash
  Trash' = Trash + f
  File' = File
}
pred restore[f : File] { ... }
pred empty { ... }

fact init { no Trash }
fact transitions {
  always (
    (some f: File | delete[f] or restore[f]) or empty
  )
}
```

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

TRASH BEHAVIOR FIXED

```
pred delete[f : File] { ... }
pred restore[f : File] { ... }
pred empty { ... }
pred do_nothing {
  Trash' = Trash
  File' = File
}

fact init { no Trash }
fact transitions {
  always (
    (some f: File | delete[f] or restore[f]) or empty or do_nothing
  )
}
```

ASSERTIONS

- In Electrum, the same first order temporal logic is used for
  - modeling
  - specification of expected properties – *assertions*
- The latter can be enclosed in named **`assert`** paragraphs

EXAMPLE ASSERTIONS

```
assert restoreAfterDelete {
  -- Every restored file was once deleted
  always (all f : File | restore[f] implies once delete[f])
}

assert deleteAll {
  -- If the trash contains all files and is emptied
  -- then no files will ever exist afterwards
  always ((File in Trash and empty) implies always no File)
}
```

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 also be explored like instances


DEMO

FIXED ASSERTION

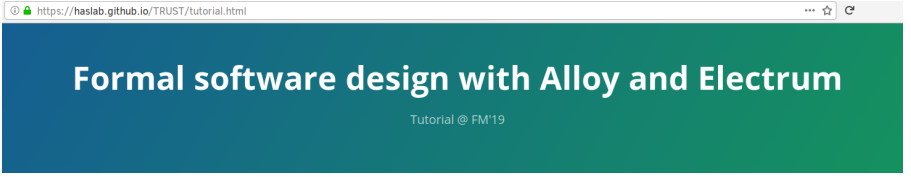
```
assert deleteAll {
  -- If the trash contains all files and is emptied
  -- then no files will ever exist afterwards
  always ((File in Trash and empty) implies after (always no File))
}
```

OUTLINE

OUTLINE OF THE TUTORIAL

Session	Time	Topic
1	9:00	Overview
	10:00	Coffee break
2	10:30	Relational logic
3	11:30	Analysis
	12:30	Lunch
4	14:00	Temporal logic
	15:30	Coffee break
5	16:00	Methodology and tips
6	17:00	Alloy4Fun

USEFUL INFO, DOWNLOAD LINKS, EXERCISES



https://haslab.github.io/TRUST/tutorial.html

EXERCISES

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

(exercises 1-3)