Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations

Nuno Macedo[1] Julien Brunel[2] David Chemouil[2]
Alcino Cunha[1] Denis Kuperberg[3]

[1]HASLab, INESC TEC and Universidade do Minho

[2]DTIM, UFTMiP, ONERA

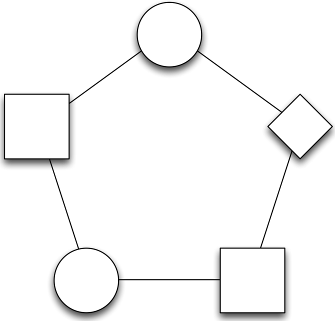
[3]TU Munich

TRUST Workshop 2016
September
Braga, Portugal

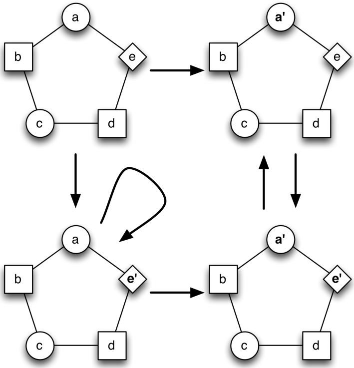
Software Design

- Trustworthy design frameworks are crucial to achieve high-assurance software
- Should provide a simple, yet flexible, formal specification **language**
- Should be accompanied by effective **tools** to support their analysis

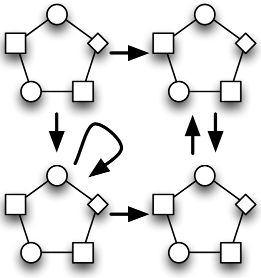
Structure



Behavior



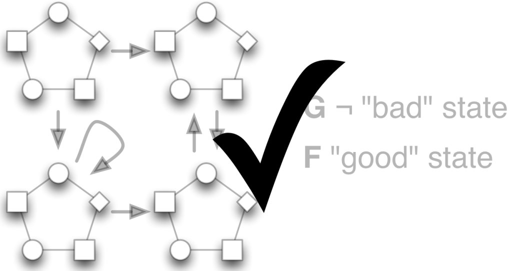
Properties



G ¬ "bad" state

F "good" state

Verification



Dynamic Systems with Rich Configurations

The language should allow:

- Clear distinction between **configurations** and **evolution**
- Configurations defined by rich **structural** constraints
- Declarative specification of the allowed **evolution** steps
- Specification of **temporal** (safety and liveness) properties

Tool support should effectively **verify** properties for every valid configuration

Contenders: Structure

Alloy + Analyzer

```
sig Process {
  succ: Process,
  toSend: Id -> Time,
  id : Id
} {
  @id in Process lone -> Id
}

fact ring {
  all p: Process | Process in p.^succ
}
```

TLA$^+$ + TLC

```
CONSTANT N
ASSUME N \in Nat /\ N > 0
VARIABLES toSend, succ, elect

Sup(R) == {r[1] : r \in R} \cup {r[2] : r \in R}

R ** T == {<<r,t>> \in Sup(R) \X Sup(T) :
            \E s \in Sup(R) \cap Sup(T) :
              (<<r,s>> \in R) /\ (<<s,t>> \in T)}

RECURSIVE TC(_)
TC(R) == IF R ** R \subseteq R THEN R
         ELSE TC(R \cup R ** R)

Rel(f) == { <<r,f[r]>> : r \in DOMAIN f }

PROCESS == 0..(N - 1)

Init == /\ succ \in [PROCESS -> PROCESS]
        /\ \A p1,p2 \in PROCESS :
             <<p1,p2>> \in TC(Rel(succ))
```

Contenders: Behavior

Alloy + Analyzer

```
sig Time {
  next_ : one Time
}
one sig Loop extends Time {}
fact { next_ = next + last->Loop }

pred init[t: Time] { ... }

pred step [t,t': Time, p: Process] {
  some i: p.toSend.t {
    p.toSend.t' = p.toSend.t - i
    p.succ.toSend.t' = p.succ.toSend.t +
      (i - prevs[p.succ.id])
  }
}

pred Progress  { ... }

fact traces {
  init
  all t:Time, t':t.next_ |
    some p: Process | step[t,t',p] || skip[t,t']
}
```

TLA$^+$ + TLC

```
Init == ...

Act(p) ==
  /\ p \in PROCESS
  /\ toSend' = [toSend EXCEPT ![succ[p]] =
        IF toSend[succ[p]] < toSend[p]
        THEN toSend[p] ELSE @]
  /\ elect' =
        IF toSend[p] = succ[p]
        THEN elect \cup {succ[p]} ELSE elect
  /\ UNCHANGED <<succ>>

Next == \E p \in PROCESS : Act(p)

Spec == /\ Init /\ [][Next]_vars
        /\ \A p \in PROCESS : WF_vars(Act(p))
```

Contenders: Properties + Verification

Alloy + Analyzer

```
assert Liveness {
  some Process && Progress =>
    some t: init.*next_ { some elect.t }
}

assert Safety {
  all t: init.*next_ { lone elect.t}
}

check Safety for 4 but 10 Time
```

language fully supported by Analyzer

bounded analysis

TLA$^+$ + TLC

```
Liveness == <>(elect /= {})
Safety ==
  \neg <>(\E i1,i2 \in elect : i1 /= i2)


SPECIFICATION Spec
PROPERTY Safety
CONSTANTS
  PROCESS = {0, 1, 2, 3}
```

language restrictions imposed by TLC

unbounded analysis

Electrum

- A **lightweight formal specification language**, inspired by Alloy and TLA that simplifies the specification of dynamic systems with rich configurations
- A bounded and an unbounded **model-checking technique** to verify such systems, i.e., whether temporal properties hold for every possible configuration

Language

- Extends Alloy with TLA features
- Time is now implicit
- Distinction between static and variable structures (configurations)
- Predicates may relate succeeding states (primed variables)
- Introduces LTL operators (X, G, F, U, R)

Example

```
sig Process {
  succ: Process,
  var toSend: set Id,
  id : Id
} {
  @id in Process lone -> Id
}

fact ring {
  all p: Process | Process in p.^succ
}

pred init [] { ... }

pred step [p: Process] {
  some i: p.toSend {
    p.toSend' = p.toSend - i
    p.succ.toSend' = p.succ.toSend +
      (i - prevs[p.succ.id])
  }
}
```

```
fact traces {
  init
  always (some p: Process | step[p] || skip)
}

assert Safety {
  always lone elect
}

assert Liveness {
  some Process && Progress =>
    eventually some elect
}

check Safety for 4
```

Example: Event Idiom

```
one var abstract sig Event {
  g: Guest
}

var abstract sig FDEvent extends Event { } {
  currentKey' = currentKey
}

var sig Checkin extends FDEvent {
  r: Room,
  k: Key
} {
  g.gKeys' = g.gKeys + k
  no FD.occupant[r]
  FD.occupant' = FD.occupant + r -> g
  FD.lastKey' = FD.lastKey ++ r -> k
  k = nextKey[FD.lastKey[r], r.keys]
  all gg: Guest - g | gg.gKeys' = gg.gKeys
}
```

Example: SPL

```
abstract sig Feature {}
one sig FIdle, FExecutive, FPark extends Feature {}

sig Product in Feature {} {
  FIdle + FPark not in this
}

sig Floor {} {
  one b: LandingButton | b.floor = this
  one b: LiftButton | b.floor = this
}

abstract sig Button { floor: one Floor }
sig LandingButton, LiftButton extends Button {}

var one sig Current in Floor {}
var lone sig Open, Up {}
var sig Pressed in Button {}

...

pred prop {
  always all f: Floor | floor.f&LiftButton in Pressed =>
    eventually (current = f && some Open) }

check { FIdle = Product => prop } for 6
```

Semantics

- Best presented through a translation into a kernel (no sig structure, no spurious operators), e.g.

Electrum

```
abstract sig A { r: some A }
var sig B,C extends A {}
```

Kernel

```
always A' = A
always (A = B + C and no B & C)

always r in A -> A
always all a: A | some a.r
```

- Variable sigs are not proper types: atoms may change identity
- Standard translation into FOLTL

Verification

- Run and check commands integrated in the specification (as Alloy)
- Scopes refer to the number of atoms in the complete life of the system
- Two backends: *Analyzer* and *nuXmv*

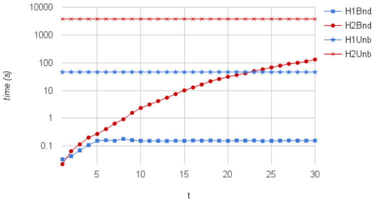
Bounded Verification

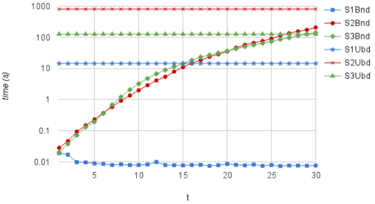
- Resulting FOLTL is converted into Alloy's FOL (cf. bounded model checking)
- Deployed over the Alloy Analyzer and its visualizer
- Iterative process to simulate minimal traces
- Allows instance iteration
- Bounded traces, improved performance

Unbounded Verification

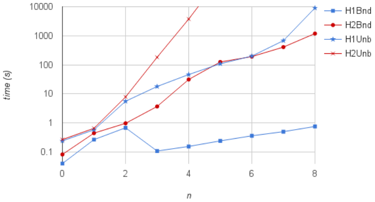
- Resulting FOLTL is converted into SMV's LTL
- Deployed over nuXmv
- Produces traces of unlimited length
- Does not allow instance iteration
- Unbounded traces, reduced performance

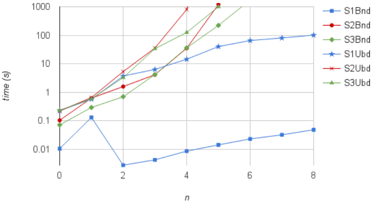
Evaluation: Increasing Trace





Evaluation: Increasing Size





Conclusions

Electrum = Structure + Behavior + Properties + Verification

- Lightweight, flexible, best aspects of Alloy and TLA$^+$
- Bounded technique suitable for early stages, unbounded for further validation

Future Work

- Improve performance by exploring non-symmetric configurations
- Improve scenario exploration functionalities