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

FORMAL SOFTWARE DESIGN WITH ALLOY AND ELECTRUM

METHODOLOGY AND TIPS

Universidade do Minho & INESC TEC

ONERA DTIS & Université fédérale de Toulouse

Carnegie Mellon University

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

THE EUROPEAN RAIL TRAFFIC MANAGEMENT SYSTEM HL3

BASIC IDEA

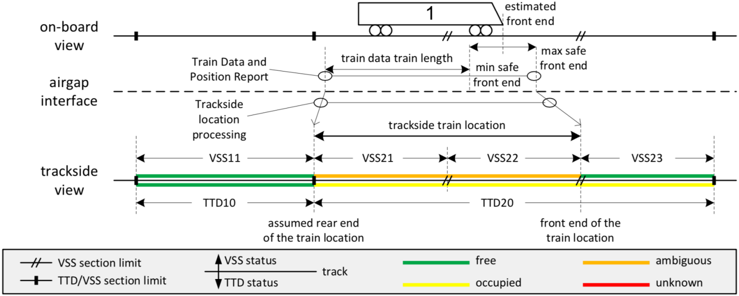
- aim: validate an *railway traffic management system* concept
- combines trackside and train reports for finer management
- specification provided, backed by operational scenarios

Challenges

- alternative track configurations
- under-specified behavior
- continuous aspects

HYBRID ERTMS/ETCS LEVEL 3

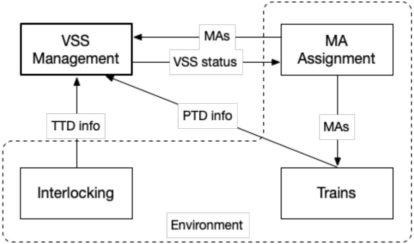
- occupancy of trackside sections determined by safe sensors (may have delays)
- occupancy of virtual sub-sections determined by train reports (communication may fail, integrity may be lost)



Hybrid ERTMS/ETCS Level 3 - Principles

HL3 — ENVIRONMENT

- train state and reporting (PTD)
- trackside sensor information (TTD)
- management authority (MA) assignment sub-system
- VSS management encoded as a state machine
- MA assignment and train reaction to it outside scope



ERTMS HL3 IN ELECTRUM

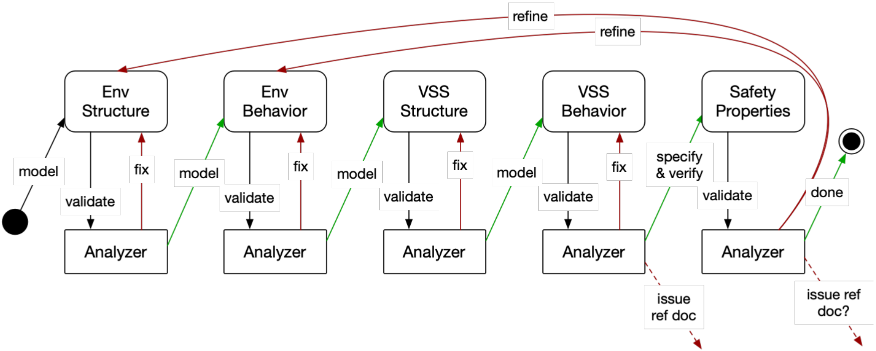
METHODOLOGY AND TIPS

- modeling
    - how to develop large models?
        - develop incrementally
    - how to model an (underspecified) environment?
        - combine explicit events with declarative temporal specifications
    - how to handle continuous aspects?
        - *sweet spot* abstractions
- validation
    - how to generate interesting scenarios?
        - use the simulator to guide exploration
        - encode specific operational scenarios *a la* unit tests
    - how to understand scenarios?
        - enrich the model with visualization-specific entities
        - define suitable visualization themes
- specification and verification
    - how to detect and deal with spurious counter-examples?
        - refine environment assumptions as needed

METHODOLOGY AND TIPS

- modeling
    - how to develop large models?
        - develop incrementally
    - how to model an (underspecified) environment?
        - combine explicit events with declarative temporal specifications
    - how to handle continuous aspects?
        - *sweet spot* abstractions
- validation
    - how to generate interesting scenarios?
        - use the simulator to guide exploration
        - encode specific operational scenarios *a la* unit tests
    - how to understand scenarios?
        - enrich the model with visualization-specific entities
        - define suitable visualization themes
- specification and verification
    - how to detect and deal with spurious counter-examples?
        - refine environment assumptions as needed

MODELING: DEVELOP INCREMENTALLY

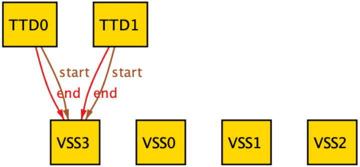


MODELING: DEVELOP INCREMENTALLY

```
open util/ordering[TTD] as D
open util/ordering[VSS] as V

sig VSS {}
sig TTD {
  start : one VSS,
  end   : one VSS
} { end.gte[start] }
```

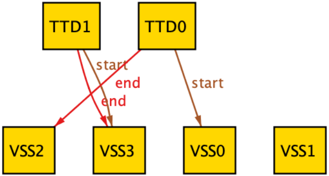
MODELING: DEVELOP INCREMENTALLY

```
open util/ordering[TTD] as D
open util/ordering[VSS] as V

sig VSS {}
sig TTD {
  start : one VSS,
  end   : one VSS
} { end.gte[start] }
```



```
run {} for 2 TTD, 4 VSS
```

MODELING: DEVELOP INCREMENTALLY

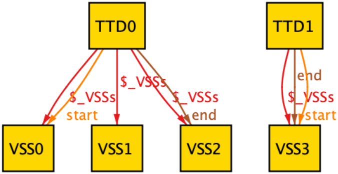


```
open util/ordering[TTD] as D
open util/ordering[VSS] as V

sig VSS {}
sig TTD {
  start : one VSS,
  end   : one VSS
} { end.gte[start] }

fact trackSections {
  all ttd:TTD-D/last | ttd.end.V/next = (ttd.D/next).start
  D/first.start = V/first and D/last.end= V/last }



run {} for 2 TTD, 4 VSS
```

METHODOLOGY AND TIPS

- modeling
  - how to develop large models?
    - develop incrementally
  - how to model an (underspecified) environment?
    - combine explicit events with declarative temporal specifications
  - how to handle continuous aspects?
    - *sweet spot* abstractions
- validation
  - how to generate interesting scenarios?
    - use the simulator to guide exploration
    - encode specific operational scenarios *a la* unit tests
  - how to understand scenarios?
    - enrich the model with visualization-specific entities
    - define suitable visualization themes
- specification and verification
  - how to detect and deal with spurious counter-examples?
    - refine environment assumptions as needed

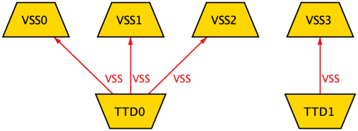
VALIDATION: VISUALIZATION-SPECIFIC ENTITIES



```
open util/ordering[TTD] as D
open util/ordering[VSS] as V

sig VSS {}
sig TTD {
  start : one VSS,
  end   : one VSS
} { end.gte[start] }

fact trackSections {
  all ttd:TTD-D/last | ttd.end.V/next = (ttd.D/next).start
  D/first.start = V/first and D/last.end= V/last }

fun _VSSs : TTD -> VSS {
  { t:TTD, v: t.start.*V/next & t.end.*(~V/next) } }

run {} for 2 TTD, 4 VSS
```

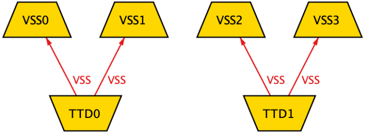
VALIDATION: THEME CUSTOMIZATION

```
open util/ordering[TTD] as D
open util/ordering[VSS] as V

sig VSS {}
sig TTD {
  start : one VSS,
  end   : one VSS
} { end.gte[start] }

fact trackSections {
  all ttd:TTD-D/last | ttd.end.V/next = (ttd.D/next).start
  D/first.start = V/first and D/last.end= V/last }

fun _VSSs : TTD -> VSS {
  { t:TTD, v: t.start.*V/next & t.end.*(~V/next) } }

run {} for 2 TTD, 4 VSS
```



alternative config?

VALIDATION: THEME CUSTOMIZATION

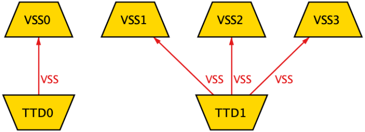


alternative config?

```
open util/ordering[TTD] as D
open util/ordering[VSS] as V

sig VSS {}
sig TTD {
  start : one VSS,
  end   : one VSS
} { end.gte[start] }

fact trackSections {
  all ttd:TTD-D/last | ttd.end.V/next = (ttd.D/next).start
  D/first.start = V/first and D/last.end= V/last }

fun _VSSs : TTD -> VSS {
  { t:TTD, v: t.start.*V/next & t.end.*(~V/next) } }

run {} for 2 TTD, 4 VSS
```

VALIDATION: THEME CUSTOMIZATION

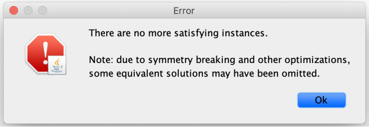
```
open util/ordering[TTD] as D
open util/ordering[VSS] as V

sig VSS {}
sig TTD {
  start : one VSS,
  end   : one VSS
} { end.gte[start] }

fact trackSections {
  all ttd:TTD-D/last | ttd.end.V/next = (ttd.D/next).start
  D/first.start = V/first and D/last.end= V/last }

fun _VSSs : TTD -> VSS {
  { t:TTD, v: t.start.*V/next & t.end.*(~V/next) } }

run {} for 2 TTD, 4 VSS
```



alternative config?

VALIDATION: THEME CUSTOMIZATION

```
open util/ordering[TTD] as D
open util/ordering[VSS] as V

sig VSS {}
sig TTD {
  start : one VSS,
  end   : one VSS
} { end.gte[start] }

fact trackSections {
  all ttd:TTD-D/last | ttd.end.V/next = (ttd.D/next).start
  D/first.start = V/first and D/last.end= V/last }

fun _VSSs : TTD -> VSS {
  { t:TTD, v: t.start.*V/next & t.end.*(~V/next) } }

run {} for 2 TTD, 4 VSS
```



Error

There are no more satisfying instances.

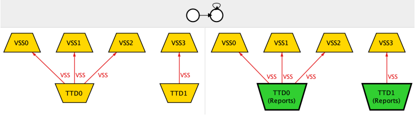
Note: due to symmetry breaking and other optimizations, some equivalent solutions may have been omitted.

alternative config?

METHODOLOGY AND TIPS

- modeling
  - how to develop large models?
    - develop incrementally
  - how to model an (underspecified) environment?
    - combine explicit events with declarative temporal specifications
  - how to handle continuous aspects?
    - *sweet spot* abstractions
- validation
  - how to generate interesting scenarios?
    - use the simulator to guide exploration
    - encode specific operational scenarios *a la* unit tests
  - how to understand scenarios?
    - enrich the model with visualization-specific entities
    - define suitable visualization themes
- specification and verification
  - how to detect and deal with spurious counter-examples?
    - refine environment assumptions as needed

MODELING: COMBINE EVENT WITH DECLARATIVE CONSTRAINTS

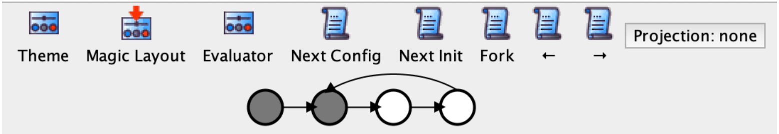


```
var sig Reports in TTD {}

fact TTDReports {
  always all t:TTD |
    t not in Reports implies t in Reports'
}

run {eventually some Reports} for 2 TTD, 4 VSS
```

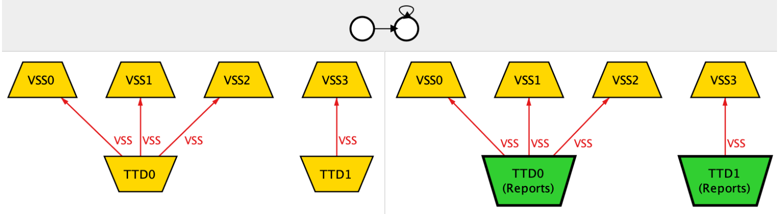
METHODOLOGY AND TIPS

- modeling
  - how to develop large models?
    - develop incrementally
  - how to model an (underspecified) environment?
    - combine explicit events with declarative temporal specifications
  - how to handle continuous aspects?
    - *sweet spot* abstractions
- validation
  - how to generate interesting scenarios?
    - use the simulator to guide exploration
    - encode specific operational scenarios *a la* unit tests
  - how to understand scenarios?
    - enrich the model with visualization-specific entities
    - define suitable visualization themes
- specification and verification
  - how to detect and deal with spurious counter-examples?
    - refine environment assumptions as needed

VALIDATION: GUIDED EXPLORATION

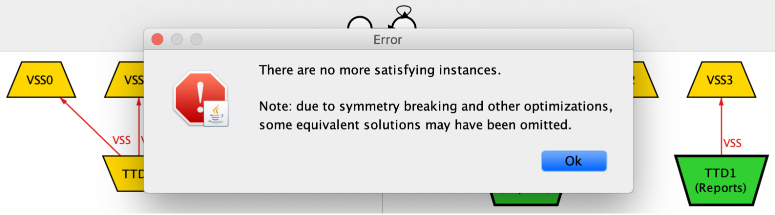


VALIDATION: GUIDED EXPLORATION



alternative transition?

VALIDATION: GUIDED EXPLORATION

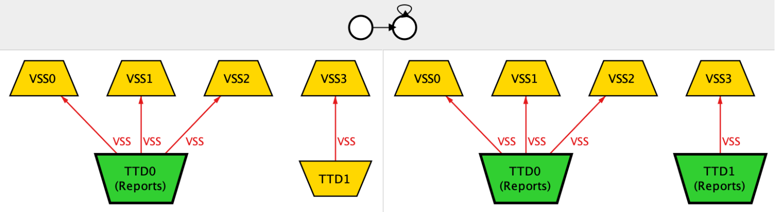


There are no more satisfying instances.

Note: due to symmetry breaking and other optimizations, some equivalent solutions may have been omitted.

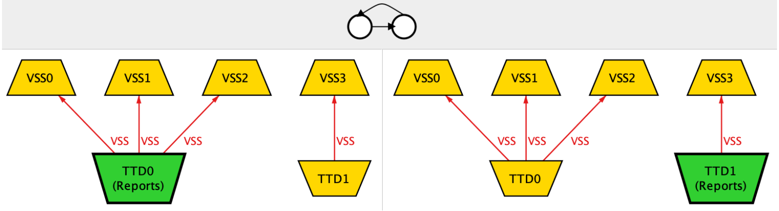
what if another initial state?

VALIDATION: GUIDED EXPLORATION



alternative transition?

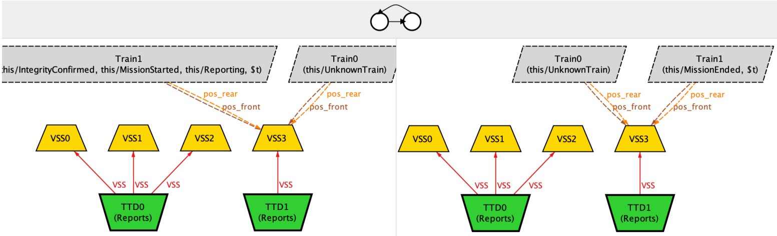
VALIDATION: GUIDED EXPLORATION



MODELING: COMBINE EVENT WITH DECLARATIVE CONSTRAINTS

```
fact TTDReports { always all t:TTD | ... }

pred move[t:Train] { ... }
pred som[t:Train] { ... }
pred eom[t:Train] { ... }
pred split[t1,t2:Train] { ... }

fact trainEvolution {
  always all t:Train |
    move[t] or som[t] or eom[t] or some t1:Train | split[t,t1] or split[t1,t]
}

run {
  some t:Train | eventually (som[t] and eventually eom[t])
} for 4 VSS, 2 TTD, 2 Train
```

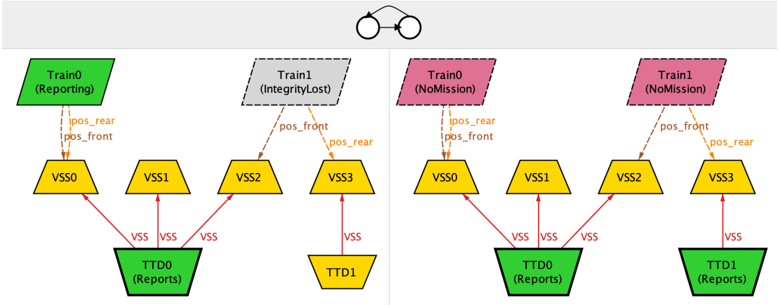
VALIDATION: VISUALIZATION-SPECIFIC ENTITIES



VALIDATION: VISUALIZATION-SPECIFIC ENTITIES

```
fun NoMission : set Train {
  MissionEnded
}
fun MissionOnly : set Train {
  MissionStarted - Reporting
}
fun ReportingOnly : set Train {
  Reporting - (IntegrityConfirmed + IntegrityLost)
}
```

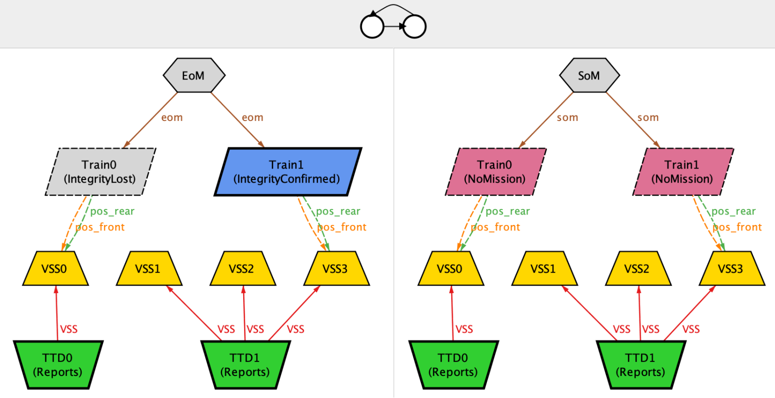
VALIDATION: VISUALIZATION-SPECIFIC ENTITIES



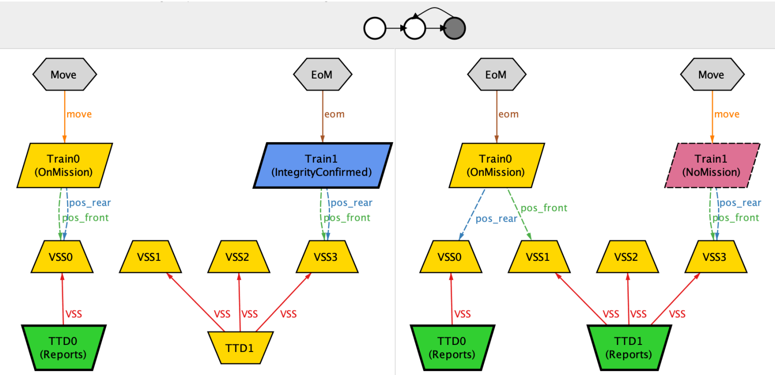
VALIDATION: VISUALIZATION-SPECIFIC ENTITIES

```
enum Event { Move, SoM, EoM, Split }

fun move : Event -> Train {
  Move -> { t:Train | move[t] }
}
fun som : Event -> Train { ... }
fun eom : Event -> Train { ... }
fun split : Event -> Train -> Train {
  Split -> { t1,t2:Train | split[t1,t2] }
}

fun events : set Event {
  (move+som+eom+split.Train).Train
}
```

VALIDATION: VISUALIZATION-SPECIFIC ENTITIES



VALIDATION: VISUALIZATION-SPECIFIC ENTITIES



METHODOLOGY AND TIPS

- modeling
  - how to develop large models?
    - develop incrementally
  - how to model an (underspecified) environment?
    - combine explicit events with declarative temporal specifications
  - how to handle continuous aspects?
    - *sweet spot* abstractions
- validation
  - how to generate interesting scenarios?
    - use the simulator to guide exploration
    - encode specific operational scenarios *a la* unit tests
  - how to understand scenarios?
    - enrich the model with visualization-specific entities
    - define suitable visualization themes
- specification and verification
  - how to detect and deal with spurious counter-examples?
    - refine environment assumptions as needed

MODELING: SWEET SPOT ABSTRACTIONS

```
var sig DiscPropRunning, DiscPropExpired in VSS {}

fun DiscPropStart : set VSS {
  { v:VSS | some t : Train |
    (v in MAs[t] and t in MuteExpired'-MuteExpired and v.state' = Unknown) or ... }
}

fun DiscPropStop : set VSS {
  { v:VSS | (all t : Train | once ((v in located[t] and eom[t]) or ...)
      implies t not in Disconnected') }
}

pred setDiscPropTimer {
  DiscPropExpired in DiscPropRunning
  no DiscPropExpired & DiscPropExpired'
  DiscPropRunning' =
    (DiscPropRunning-DiscPropExpired-DiscPropStop)+DiscPropStart
}
```

METHODOLOGY AND TIPS

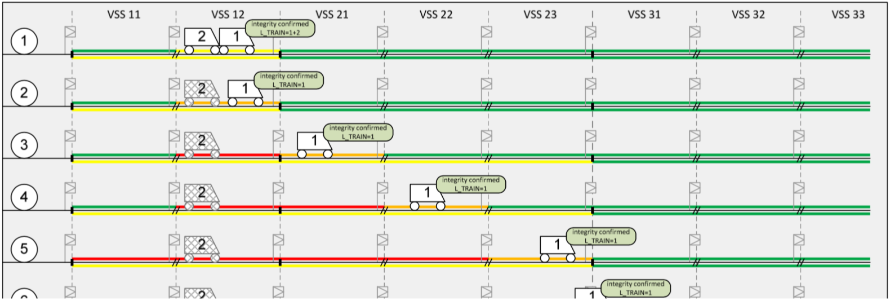
- modeling
  - how to develop large models?
    - develop incrementally
  - how to model an (underspecified) environment?
    - combine explicit events with declarative temporal specifications
  - how to handle continuous aspects?
    - *sweet spot* abstractions
- validation
  - how to generate interesting scenarios?
    - use the simulator to guide exploration
    - encode specific operational scenarios *a la* unit tests
  - how to understand scenarios?
    - enrich the model with visualization-specific entities
    - define suitable visualization themes
- specification and verification
  - how to detect and deal with spurious counter-examples?
    - refine environment assumptions as needed

VALIDATION: ENCODING SCENARIOS

```
run {
  some disj t1, t2 : Train, v : VSS {
    eventually (v in located[t1] = v;v in located[t2])
    always Train in MissionStarted }
} for 4..6 Time, 2 Train, 3 TTD, 8 VSS
```

HL3 OPERATIONAL SCENARIOS

- environment evolution restricted
- validate whether VSS system and timers act as expected

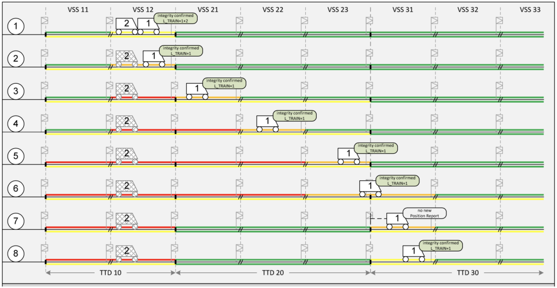

VSS 11
VSS 12
VSS 21
VSS 22
VSS 23
VSS 31
VSS 32
VSS 33
1
2
3
4
5
integrity confirmed L_TRAIN=1+2
integrity confirmed L_TRAIN=1

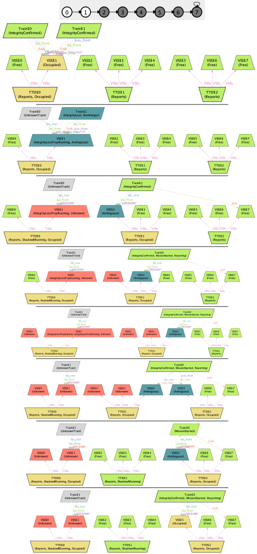
Hybrid ERTMS/ETCS Level 3 - Principles

OPERATIONAL SCENARIO #2

```
pred S2env { let v11 = V/first, v12 = v11.next, v21 = v12.next ... |
  some disj t1,t2:Train {
    v12 in parent[first].end and v31 in parent[last].start
    always TTD = Reports
    t1.pos = v12;t1.pos = v12;t1.pos = v21;...
    always t2.pos = v12
    split[t1,t2]
    t1 in IntgrtyConfirmed;t1 not in IntgrtyConfirmed;...
    ... } }

pred S2ok { let v11 = V/first, v12 = v11.next, v21 = v12.next ... |
  eventually always {
    (v11+v12).state = Unknown
    v31.state = Occupied
    v21+v22+v23+v32+v33).state = Free }
  after (v12 = IntgrtyLossPropRunning;v12 = IntgrtyLossPropRunning) }

run { S2 and S2ok } for exactly 2 Train, exactly 3 TTD, exactly 8 VSS, exactly 8 Time
```

OPERATIONAL SCENARIO #2





HL3 FOUND ISSUES

- inconsistencies between VSS system description and scenarios
  - state machine transition conditions vs. behavior in scenarios (*fixed in current version*)
  - timer behavior (indefinite expiration) vs. behavior in scenarios (*fixed in current version*)
  - timer stop conditions vs. behavior in the scenarios
- possible issues
  - ambiguous nomenclature (*fixed in current version*)
  - state machine does not stabilize
  - missing timer starts in scenarios

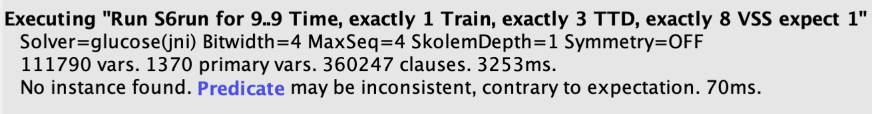
VALIDATION: ENCODING SCENARIOS

```
fun DisconnectPropStop : set VSS {
  ...
  v.state' != v.state and v.state' in Occupied+Ambiguous+Free
  ...
}

pred S6ok {
  ...
  after after (v12 = DisconnectPropRunning;v12 = DisconnectPropRunning)
  ...
}
```

Issue

Reference behavior inconsistent with scenarios

VALIDATION: ENCODING SCENARIOS

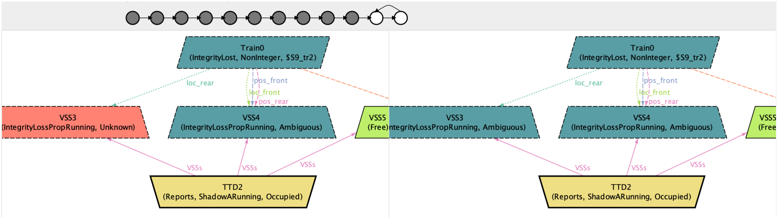


```
Executing "Run S6run for 9..9 Time, exactly 1 Train, exactly 3 TTD, exactly 8 VSS expect 1"
   Solver=glucose(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF
   111790 vars. 1370 primary vars. 360247 clauses. 3253ms.
   No instance found. Predicate may be inconsistent, contrary to expectation. 70ms.
```

Issue

Reference behavior inconsistent with scenarios

VALIDATION: GUIDED EXPLORATION

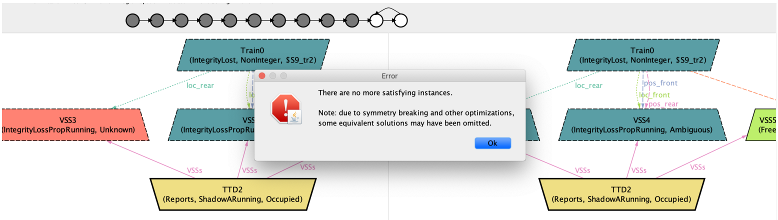

Train0
(IntegrityLost, NonInteger, $S9_tr2)
loc_rear
pos_front
loc_front
pos_rear
VSS3
(IntegrityLossPropRunning, Unknown)
VSS4
(IntegrityLossPropRunning, Ambiguous)
VSS5
(Free)
VSSs
VSSs
VSSs
TTD2
(Reports, ShadowARunning, Occupied)
Train0
(IntegrityLost, NonInteger, $S9_tr2)
loc_rear
pos_front
loc_front
pos_rear
VSS3
ntegrityLossPropRunning, Ambiguous)
VSS4
(IntegrityLossPropRunning, Ambiguous)
VSS5
(Free
VSSs
VSSs
VSSs
TTD2
(Reports, ShadowARunning, Occupied)

is there an alternative transition?

Issue

State machine does not stabilize

VALIDATION: GUIDED EXPLORATION



Error

There are no more satisfying instances.

Note: due to symmetry breaking and other optimizations, some equivalent solutions may have been omitted.

Ok
Train0
(IntegrityLost, NonInteger, $S9_tr2)
loc_rear
VSS3
(IntegrityLossPropRunning, Unknown)
VSS
(IntegrityLossPropRu
VSSs
TTD2
(Reports, ShadowARunning, Occupied)
pos_front
loc_front
pos_rear
VSS4
(IntegrityLossPropRunning, Ambiguous)
VSS5
(Free

Issue

State machine does not stabilize

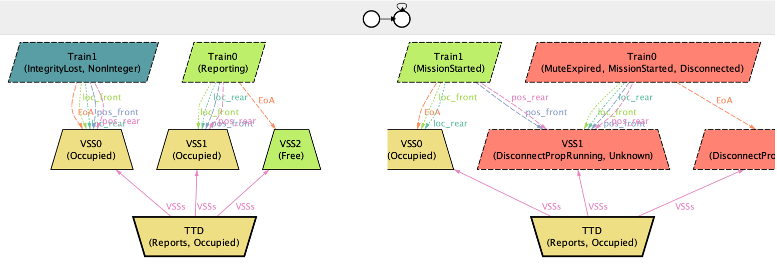
METHODOLOGY AND TIPS

- modeling
    - how to develop large models?
        - develop incrementally
    - how to model an (underspecified) environment?
        - combine explicit events with declarative temporal specifications
    - how to handle continuous aspects?
        - *sweet spot* abstractions
- validation
    - how to generate interesting scenarios?
        - use the simulator to guide exploration
        - encode specific operational scenarios *a la* unit tests
    - how to understand scenarios?
        - enrich the model with visualization-specific entities
        - define suitable visualization themes
- specification and verification
    - how to detect and deal with spurious counter-examples?
        - refine environment assumptions as needed

HL3 SAFETY PROPERTIES

```
pred noCollisions {
  no disj t1,t2:Train | some t1.pos&t2.pos
}

assert no_collisions {
  init implies always noCollisions
}

check no_collisions
for 10 Time, 8 VSS, 3 TTD, 3 Train
```

HL3 SAFETY PROPERTIES



SPECIFICATION AND VERIFICATION: REFINE ENVIRONMENT

```
assert no_collisions {
  (init and always (strictMove and instTimers)) implies
    always noCollisions
}
```

Caveat

- trial and error manual process, not validated
- do not hold for all operational scenarios

HL3 LIVENESS PROPERTIES

```
assert liveness {
    eventually some t:Train | last in located[t]
}
```

LESSONS LEARNED

- in general more readable and elegant than Alloy (although patterns that refer to concrete time instants may become more complex)
- structural freedom (and limited module system) undermines maintainability
- concrete scenarios are burdensome to encode (new op `;`, finer **`Time`** scopes)

STTT 2019, https://doi.org/10.1007/s10009-019-00540-4

EXERCISES

EXERCISES

Try Exercise #6:

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