Event-Driven Temporal Logic for Control Software Requirements

Vladimir Zyubin, Igor Anureev, Natalia Garanina, Sergey Staroletov, Anastasia Getmanova, Anna Gnezdilova, Alexandra Grivtsova

Institute of Automation and Electrometry

Institute of Informatics Systems

Novosibirsk State University

Altay State University

Novosibirsk

Russia

Motivation: Client-Contractor Problem

Safety-critical Control Software and Formal Methods

- Problems
  - the size of systems implies a high price of using formal methods
  - the conceptual divergence in requirements understanding

- Control software development: client-contractor paradigm
  - the clients think in terms of events, timeouts, processes and states
    - the contractors: the programming languages they use
  - the clients: do not care about the internal structure of control software
    - the contractors: do not care about the internal structure of the plant
  - the clients already implicitly assume a control algorithm by design
    - the contractors must reveal and implement this hidden algorithm

- How to easy formulate a complete and correct set of requirements

Motivation: General Principles of Specification

A requirements specification should be:

1. *user-friendly*
   - concepts of events and reactions
2. *independent of control software design and implementation*
   - the black box principle
     - use just terms of inputs and outputs
       - no inner structure of software/hardware
3. *strict*
   - formal semantics
4. *universal*
   - not orientated towards any particular verification technique

Motivation: Known Solutions

- A pattern-restricted natural language (e. g. ISO/IEC/IEEE 29148)
- Domain-oriented (FSM-based) languages
- Graphic notations (UML, Statecharts)
- Formal requirement pattern languages (RSL, RSML)

- Do not meet all the requirements for requirements
  1. *user-friendly*
  2. *independent (black box)*
  3. *strict*
  4. *universal*

EDTL Requirements: Definition

- *EDTL requirement* is a tuple:

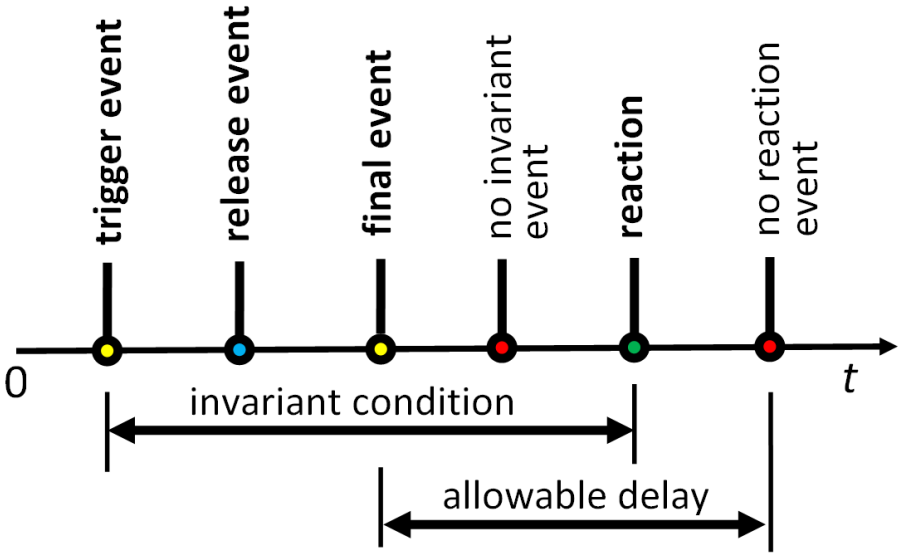
$$\mathcal{R} = (\textbf{trigger}, \textbf{invariant}, \textbf{final}, \textbf{delay}, \textbf{reaction}, \textbf{release})$$

Attribute	Description
trigger	the starting trigger event of the requirement
invariant	a condition; holds until the reaction or release
final	the trigger for ending event of the requirement; wait for the reaction
delay	the possible delay for the reaction since the final
reaction	the ending event of the requirement
release	cancels the requirement statements

EDTL Requirements: Time Ordering of Events

- *EDTL requirement* is a tuple:

$$\mathcal{R} = (\textbf{trigger}, \textbf{invariant}, \textbf{final}, \textbf{delay}, \textbf{reaction}, \textbf{release})$$



Following each trigger event, the invariant must hold true until either a release event or a final event. The invariant must also hold true after final event till either the release event or a reaction, and besides the reaction must take place within the specified allowable delay from the final event.

EDTL Requirements Syntax: Types and Terms

Types

- integers
- floating points
- Boolean
- time: 1h, 1s, etc.

EDTL-terms

- c – a constant of type T
- v – a variable of type T
- $f(u_1, \ldots, u_n)$ – a function
  - u_i – a term
  - f can be a standard arithmetic operation or relation, Boolean or C-like bitwise operation

EDTL Requirements Syntax: Formulas

EDTL formulas

- Let φ and ψ be EDTL formulas. Then:
  - ETDL term of type bool is an atomic EDTL formula
  - $\neg\varphi$ is the negation
  - $\varphi \wedge \psi$ is the conjunction
  - $\varphi \vee \psi$ is the disjunction
  - $\backslash\varphi$ is the falling edge: the value of φ changes from *false* to *true*
  - $/\varphi$ is the rising edge: the value of φ changes from *true* to *false*
  - $\_\varphi$ is low steady-state: the value of φ remains equal to *false*
  - $\sim\varphi$ is high steady-state: the value of φ remains equal to *true*

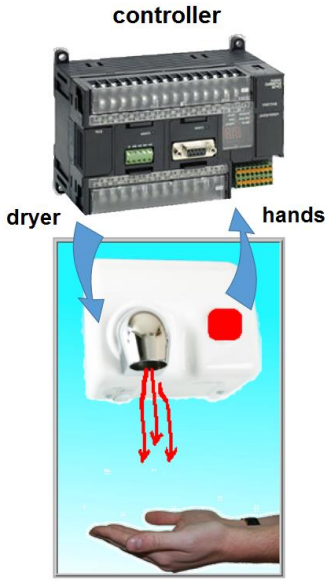
EDTL Requirements: the Hand Dryer Example

1. If the dryer is on (*D*, trigger), then it turns off (*¬D*, reaction) after no hands (*\H*, trigger) are present for 1 second

trigger	release	final	delay	invariant	reaction
\H ∧ D	H	passed 1s	*true*	D	¬D

2. If the dryer was not turned on and hands appeared (*/H ∧ ¬D* , trigger), it will turn on (*D*, reaction) ASAP (*true*, final)

trigger	release	final	delay	invariant	reaction
/H ∧ ¬D	*false*	*true*	*true*	*true*	D



EDTL Requirements: Progress

1. Automata semantics
2. FOL semantics
3. LTL semantics
4. Bounded checking algorithm
5. Consistency checking
6. Automatic translation to LTL
7. Semantic classification
8. NL representation
9. Automatic generation of dynamic verifier
10. The corpus of EDTL requirements
11. Model checking (the idea)

Semantics of EDTL Requirements

EDTL Requirements: Automata Semantics

EDTL requirement $\mathcal{R}$ is *satisfied* in a control system C iff Buchi automata $A_{\mathcal{R}}$ accepts serial evaluations of EDTL attributes in all behaviors of C.

$A_{\mathcal{R}} = (Q, \Sigma, \delta, q_0, F)$

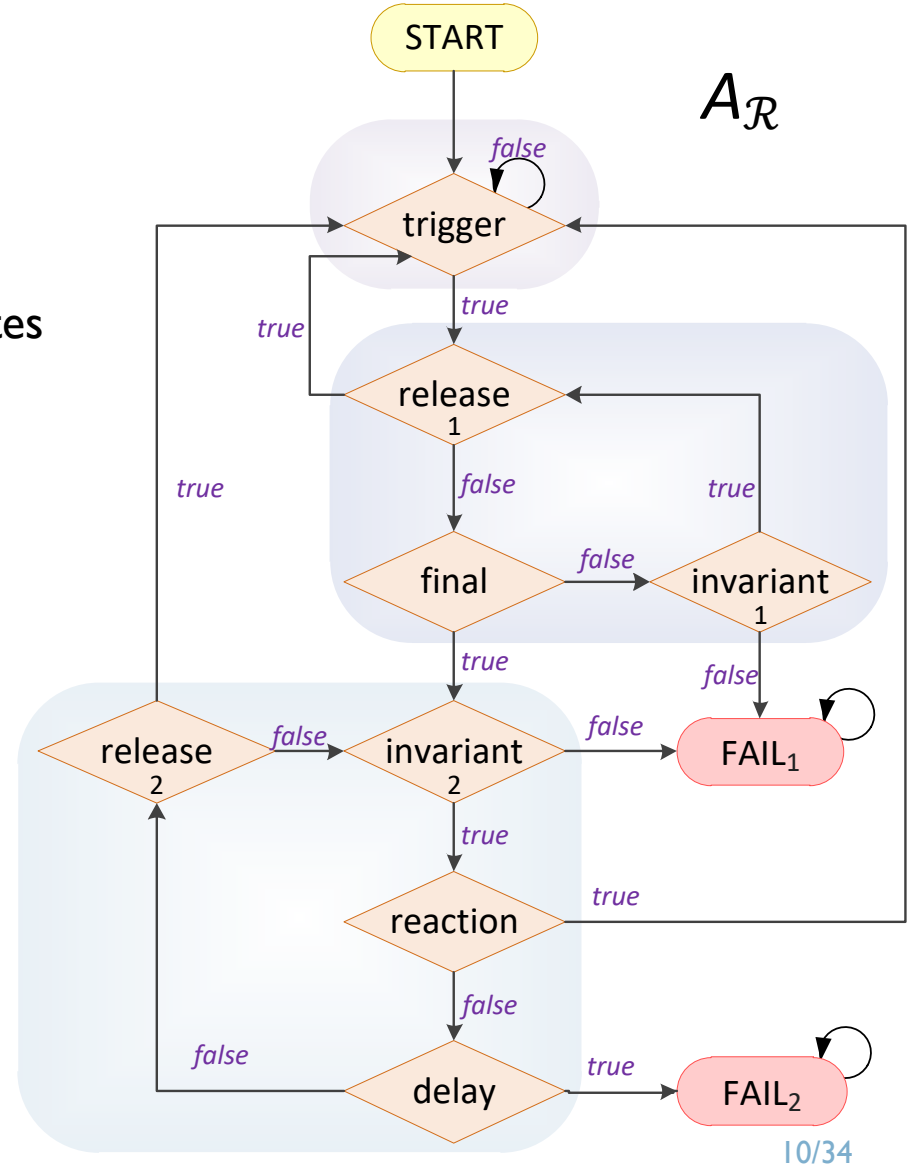
Q – {START, $FAIL_1$, $FAIL_2$, trigger, $invariant_1$, $invariant_2$, final, delay, reaction, $release_1$, $release_2$}

Σ – {*true*, *false*}

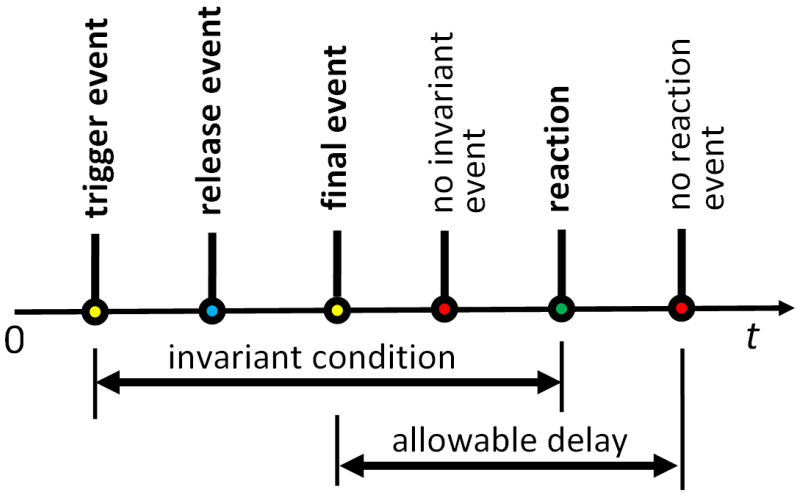
δ – trigger × *true* → $release_1$, etc.

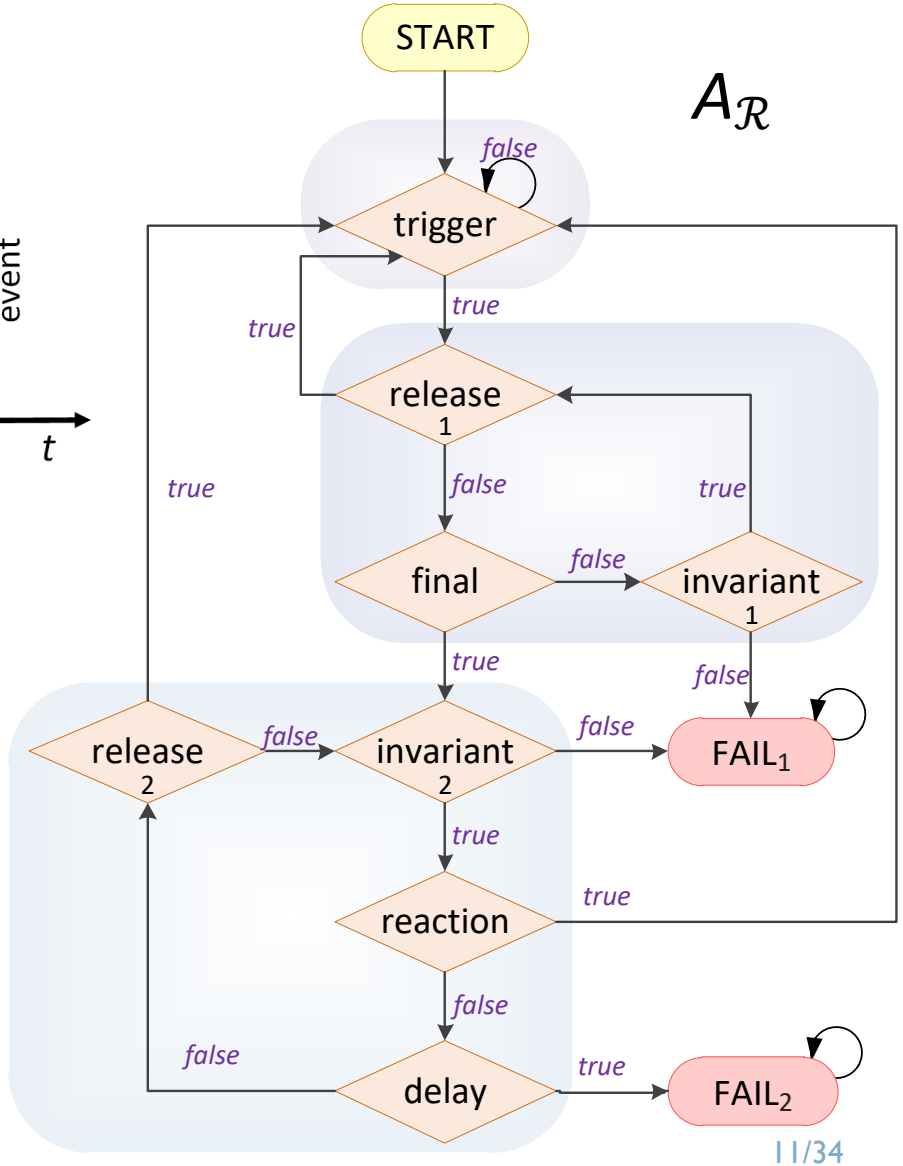
q_0 – START

F – $Q \backslash \{FAIL_1, FAIL_2\}$



EDTL Requirements: Automata Semantics

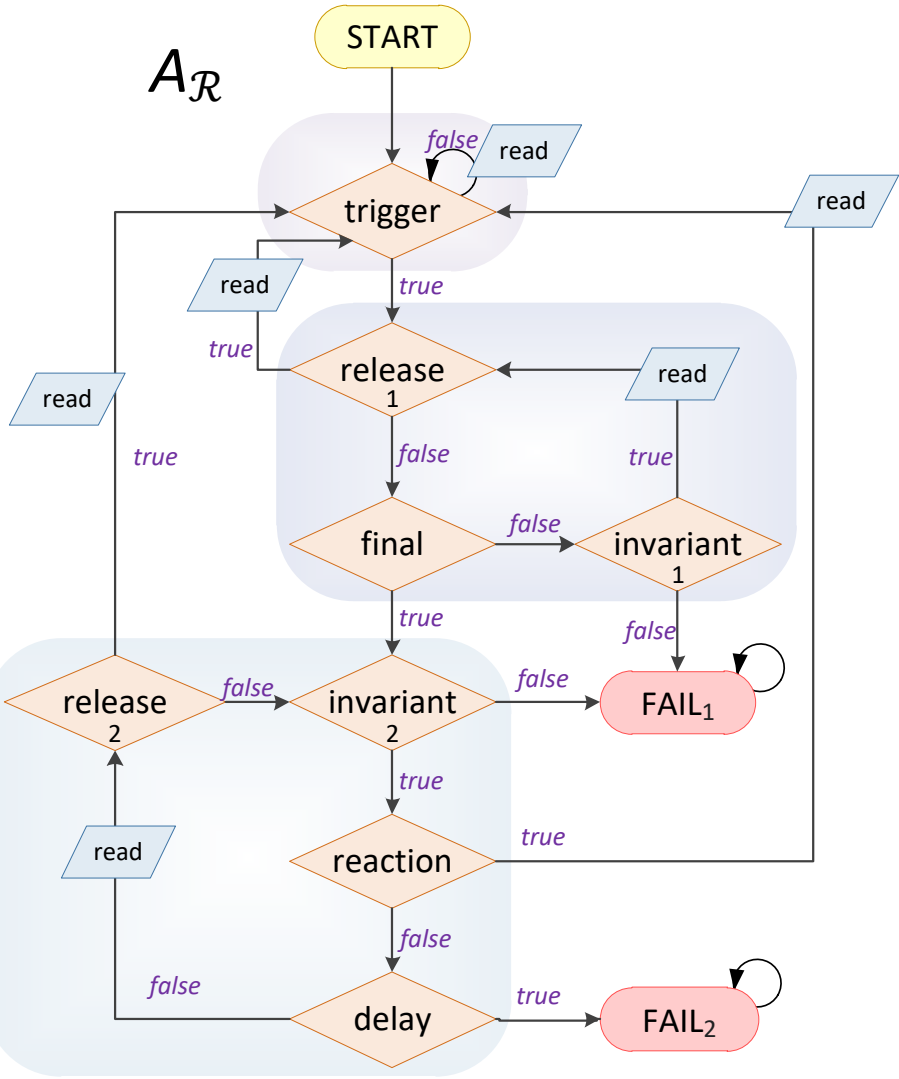




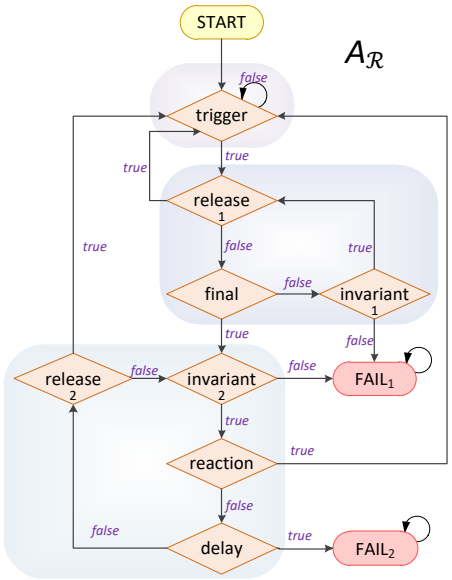
EDTL Requirements: Automata Semantics

At "read" marks the oracle

- takes new input values and
- evaluates $A_{\mathcal{R}}$ states (true/false)



EDTL Requirements: FOL semantics



- EDTL requirement $\mathcal{R}$ is satisfied in a control system C iff the following FOL formula $F_{\mathcal{R}}$ is true for every initial path π^0:

$$F_{\mathcal{R}} = \forall \pi^0 \in CS\ \forall t \in [1, +\infty)\ (\ value(trigger, \pi^0, T, 0) \Rightarrow (\ \exists F \in [T, +\infty) \exists R \in (F, +\infty)$$

$$value(final, \pi^0, F, T) \wedge value(invariant, \pi^0, F, T) \wedge value(reaction, \pi^0, R, F) \wedge$$

$$\forall i \in [T, F)\ (\neg value(release, \pi^0, i, T) \wedge value(invariant, \pi^0, i, T) \wedge \neg value(final, \pi^0, i, T)) \wedge$$

$$\forall i \in [F, R)\ (\neg value(release, \pi^0, i, T) \wedge value(invariant, \pi^0, i, T) \wedge$$

$$\neg delay(reaction, \pi^0, i, F) \wedge \neg value(reaction, \pi^0, i, F))\ \vee$$

$$\exists F \in [T, +\infty)$$

$$value(final, \pi^0, F, T) \wedge value(invariant, \pi^0, F, T) \wedge$$

$$\forall i \in [T, F)\ (\neg value(release, \pi^0, i, T) \wedge value(invariant, \pi^0, i, T) \wedge \neg value(final, \pi^0, i, T)) \wedge$$

$$\forall i \in [F, +\infty)\ (\forall j \in [F, i]\ (\neg value(release, \pi^0, j, F)) \Rightarrow$$

$$\forall j \in [F, i](value(invariant, \pi^0, j, T) \wedge \neg delay(reaction, \pi^0, i, F) \wedge \neg value(reaction, \pi^0, j+1, F)))\ \vee$$

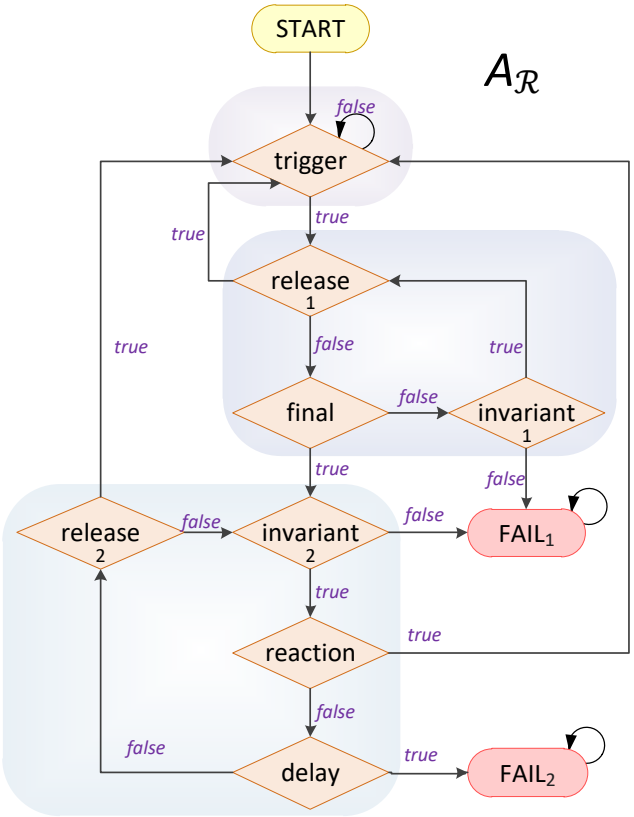
$$\forall i \in [T, +\infty)\ (\forall j \in [T, i]\ (\neg value(release, \pi^0, j, F)) \Rightarrow$$

$$\forall j \in [T, i]\ (value(invariant, \pi^0, j, T) \wedge \neg value(final, \pi^0, j, T)))))$$

The Bounded Checking Algorithm

```
bool take (struct req, array pp) {
    for (i = 0, i < n, i++)
        if !check (req, pp[i]) return false;
    return true;
}
bool check (struct req, array p) {
 trig = 1;
 while (trig < len) {
  if (value(req.trigger, p, trig, 0) {
   if (value(req.release, p, trig, trig)) goto checked;
   fin = trig
   while (!value(req.final, p, fin, trig)) {
    if (value(req.release, p, fin, trig)) goto checked;
    if (!value(req.invariant, p, fin, trig)) return false;
    fin++;
    if (fin == len) goto checked;
   }
   del = fin;
   while (!value(req.delay, p, del, fin) &&
          !value(req.reaction, p, del + 1, fin)) {
    if (value(req.release, p, del, trig)) goto checked;
    if (!value(req.invariant, p, del, fin)) return false;
    del++;
    if (del == len) goto checked;
   }
   if (!value(req.release, p, del, trig) &&
       value(req.delay, p, del, fin) &&
       !value(req.invariant, p, del, fin)) return false;
  }
  checked: trig++;
 }
 return true;
}
```

- Follows FOL formula $F_{\mathcal{R}}$ (old vers.)
  - Uses selected bounded paths



- https://doi.org/10.5281/zenodo.4445663

EDTL Requirements: LTL semantics

EDTL requirement $\mathcal{R}$ is satisfied in a control system *C* iff

LTL formula $\Phi_{\mathcal{R}}$ is satisfied in M_C for every initial path:

$\Phi_{\mathcal{R}}$ =

G(trigger ∧ ¬release → invariant ∧ (**G**(invariant ∧ ¬final) ∨

(invariant ∧ ¬final **U** release ∨ (final ∧

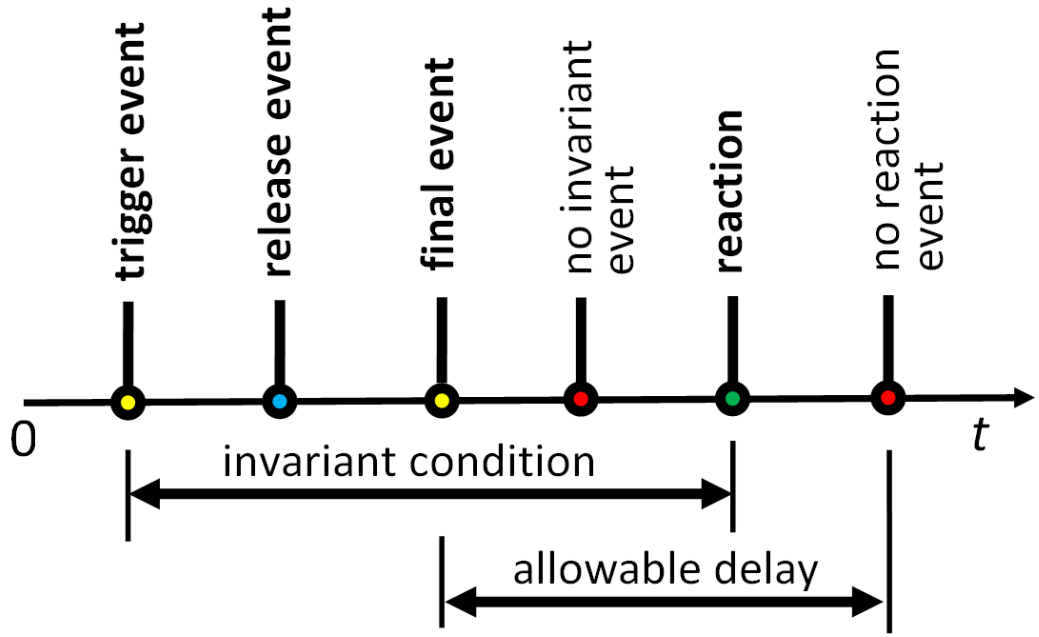
(invariant ∧ ¬delay **U** (release ∨ (invariant ∧ reaction))))))).

- M_C is a Kripke structure for control system C
- Its initial path starts from an initial state

EDTL Requirements: LTL semantics

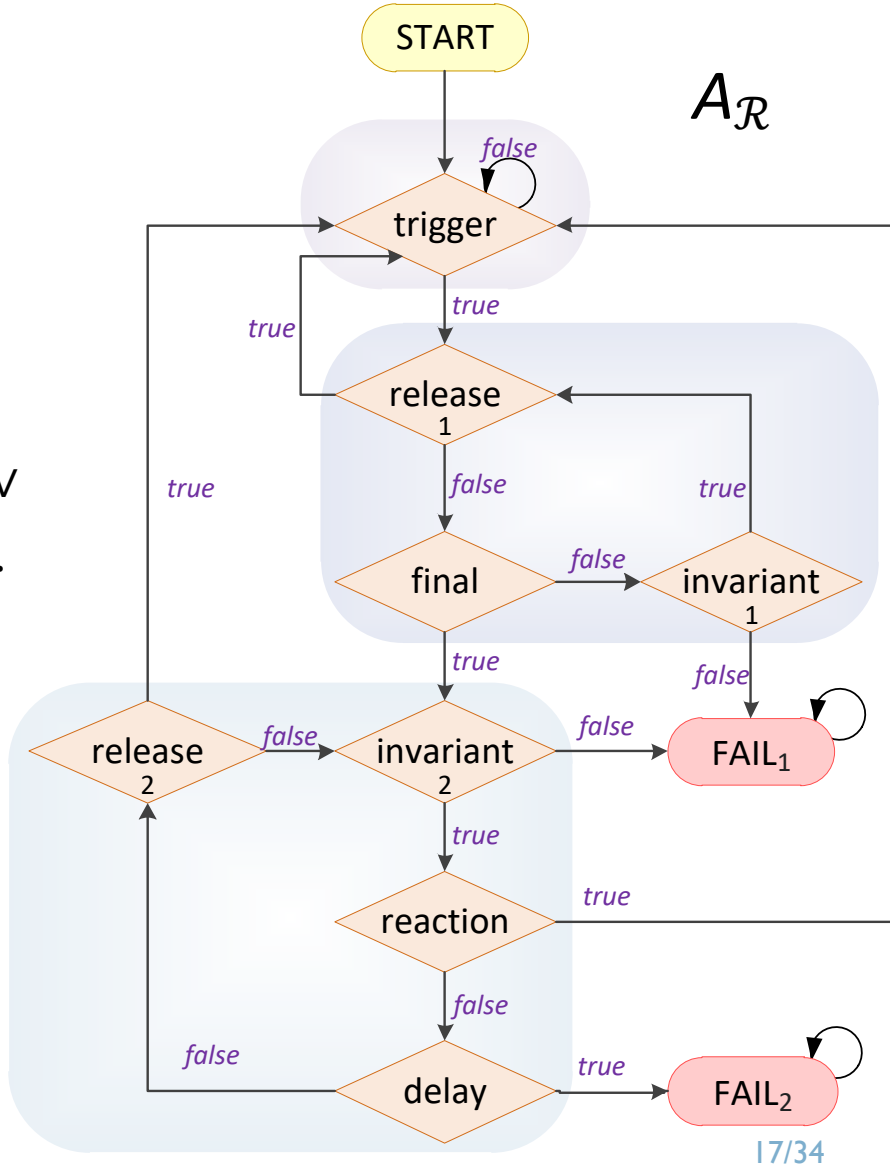
$\Phi_{\mathcal{R}}$ =

G(trigger ∧ ¬release → invariant ∧ (**G**(invariant ∧ ¬final) ∨
(invariant ∧ ¬final **U** release ∨ (final ∧
(invariant ∧ ¬delay **U** (release ∨ (invariant ∧ reaction))))))).



EDTL Requirements: LTL semantics

G(trigger ∧ ¬release → invariant ∧
(**G**(invariant ∧ ¬final) ∨
(invariant ∧ ¬final **U** release ∨
(final ∧ (invariant ∧ ¬delay **U** (release ∨
(invariant ∧ reaction))))))).



Consistency of EDTL Requirements

Consistency: Definitions

$\Phi_{\mathcal{R}}$ = **G**(trigger ∧ ¬release → invariant ∧ (**G**(invariant ∧ ¬final) ∨
(invariant ∧ ¬final **U** release ∨ (final ∧
(invariant ∧ ¬delay **U** (release ∨ (invariant ∧ reaction))))))).

Satisfiability of EDTL-requirements

- Requirement $\mathcal{R}$ is *satisfiable* iff there exists a Kripke structure $M_{\mathcal{R}}$ that for every initial path π:

$$M_{\mathcal{R}}, \pi \vDash \Phi_{\mathcal{R}}.$$

- This $M_{\mathcal{R}}$ is a model for $\mathcal{R}$: $M_{\mathcal{R}} \vDash \Phi_{\mathcal{R}}$

The checking inconsistency problem for EDTL requirements

- Requirement $\mathcal{R}'$ is *inconsistent* with satisfiable requirement $\mathcal{R}$ iff in every model $M_{\mathcal{R}}$ there exists an initial path π of $M_{\mathcal{R}}$:

$$M_{\mathcal{R}}, \pi \nvDash \Phi_{\mathcal{R}} \wedge \Phi_{\mathcal{R}'}$$

- The *checking inconsistency problem* for EDTL requirements is to check if two EDTL requirements are inconsistent.

Consistency: Method

$\Phi_{\mathcal{R}}$ = **G**(trigger ∧ ¬release → invariant ∧ (**G**(invariant ∧ ¬final) ∨
(invariant ∧ ¬final **U** release ∨ (final ∧
(invariant ∧ ¬delay **U** (release ∨ (invariant ∧ reaction)))))))

$\Phi_{\mathcal{R}}$ = **G**(tr → Ψ) and $\Phi_{\mathcal{R}'}$ = **G**(tr' → Ψ').

To check $M_{\mathcal{R}}$, π ⊭ $\Phi_{\mathcal{R}} \wedge \Phi_{\mathcal{R}'}$

¬($\Phi_{\mathcal{R}} \wedge \Phi_{\mathcal{R}'}$) ∧ (tr → tr') ⇒ ¬**G**(tr → (Ψ ∧ Ψ'))

Assumptions

- tr → tr'
- $M_{\mathcal{R}} \models \Phi_{\mathcal{R}}$

Method

- Find predicates on EDTL attributes of $\mathcal{R}$ and $\mathcal{R}'$ which is true iff $\mathcal{R}$ and $\mathcal{R}'$ are *inconsistent*.
- Function *Compare* : *Req* ×*Req* → {consistent, inconsistent, unknown}

Consistency: Algorithm

The main algorithm *Consistency_Checker*

- Input: a set of EDTL requirements Reqs
- Output: the lists of inconsistent, consistent and undefined requirements
- Complexity: quadratic w.r.t. the size of Reqs
- Call: function *Decide*

The function *Decide*

- Input: EDTL requirements $\mathcal{R}$ and $\mathcal{R}$'
- Output: {consistent, inconsistent, unknown}
- Complexity: constant
- Call: function *imply*, function *Compute_semantics*, function *Compare*

Consistency: Algorithm

The function *imply*

- Input: EDTL formulas f and f'
- Output: {true, false}
- Complexity: exponential w.r.t. the size of f and f'

The function *Compute_semantics*

- Input: EDTL requirements $\mathcal{R}$ and $\mathcal{R}$'
- Output: {true, false, other}
- Complexity: linear w.r.t. the size of $\mathcal{R}$ and $\mathcal{R}$'

The function *Compare*

- Input: EDTL requirements $\mathcal{R}$ and $\mathcal{R}$'
- Output: {consistent, inconsistent, unknown}
- Complexity: exponential w.r.t. the size of attributes of $\mathcal{R}$ and $\mathcal{R}$'

Consistency: Algorithm

Theorem

- There exists the algorithm partially solving the checking inconsistency problem for EDTL requirements which takes *quadratic time* w.r.t. the size of the set of requirements and *exponential time* w.r.t. the size of the requirements' attributes.

Let the size of every EDTL attribute of each EDTL requirement be *a*

Standard automata-based satisfiability checking algorithms for LTL formula φ

- exponential time $T_s(\varphi)$ w.r.t. the size of φ
- $T_s(\Phi_{\mathcal{R}} \wedge \Phi_{\mathcal{R}'}) \geq 2^{20a}$

The most expensive function *Decide*

- the time complexity $T_d(\mathcal{R}, \mathcal{R}') = T_{imply} + T_{compare}$
- $T_d(\mathcal{R}, \mathcal{R}') \leq 2^{8a}$

Simplification and Classification of EDTL Requirements

Simplification and Classification: the Idea

- Simultaneous supply of signals "Up" and "Down" to the elevator motor is prohibited.

trigger	release	final	delay	invariant	reaction
true	*false*	*true*	*true*	¬(Up ∧ Down)	*true*

- LTL semantics:
  - **G**(true ∧ ¬false⟶ ¬(Up ∧ Down) ∧ (**G**(¬(Up ∧ Down) ∧ ¬true) ∨ (¬(Up ∧ Down) ∧ ¬true) **U** (false ∨ (true ∧ (¬(Up ∧ Down) ∧ ¬true) **U** (false ∨ true ∧ ¬(Up ∧ Down))))))

- Reduced LTL semantics:
  - **G**(¬(Up ∧ Down))

Simplification: the Rules

Standard	Special
	$\varphi \vee \mathbf{F}(\varphi) \equiv \mathbf{F}(\varphi)$
$\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$	$\varphi \vee \mathbf{F}(\varphi \vee \psi) \equiv \mathbf{F}(\varphi \vee \psi)$
…	$\mathbf{G}(\neg\varphi) \vee \mathbf{F}(\varphi) \equiv \text{true}$
$\varphi \wedge \mathbf{G}(\varphi \wedge \psi) \equiv \mathbf{G}(\varphi \wedge \psi)$	$\varphi \wedge (\psi \mathbf{U} \varphi) \equiv \varphi$
	$\varphi \wedge (\varphi \mathbf{U} \psi) \equiv \varphi \mathbf{U} \psi$
	$\varphi \vee (\psi \mathbf{U} \varphi) \equiv \psi \mathbf{U} \varphi$

Simplification: the Algorithm

```
EDTL2LTL(trig, rel, fin, del, inv, rea) = (
  let f0 = con(inv, rea);
  let f1 = dis(rel, f0);
  let f2 = con(inv, no(del));
  let f3 = Until(f2, f1);
  let f4 = con(fin, f3);
  let f5 = dis(rel, f4);
  let f6 = con(inv, no(fin));
  let f7 = Until(f6, f5);
  let f8 = Globally(inv, no(fin));
  let f9 = dis(f8, f7);
  let f10 = con(inv, f9);
  let f11 = con(trig, no(rel));
  let f12 = impl(f11, f10);
  Globally(f12)
)
```

```
Until(a, b) = match(a, b) with
  _, False => False |
  False, f => f |
  True, f:NonConst => Future(f) |
  f, True => True
  f, f => f |
  not(f), f => F(f) |
  and(f, g), or(f, h) => or(f, h) |
  _, _ => U(a, b);
```

φ **U** false = false

false **U** φ = φ

true **U** φ = **F** φ

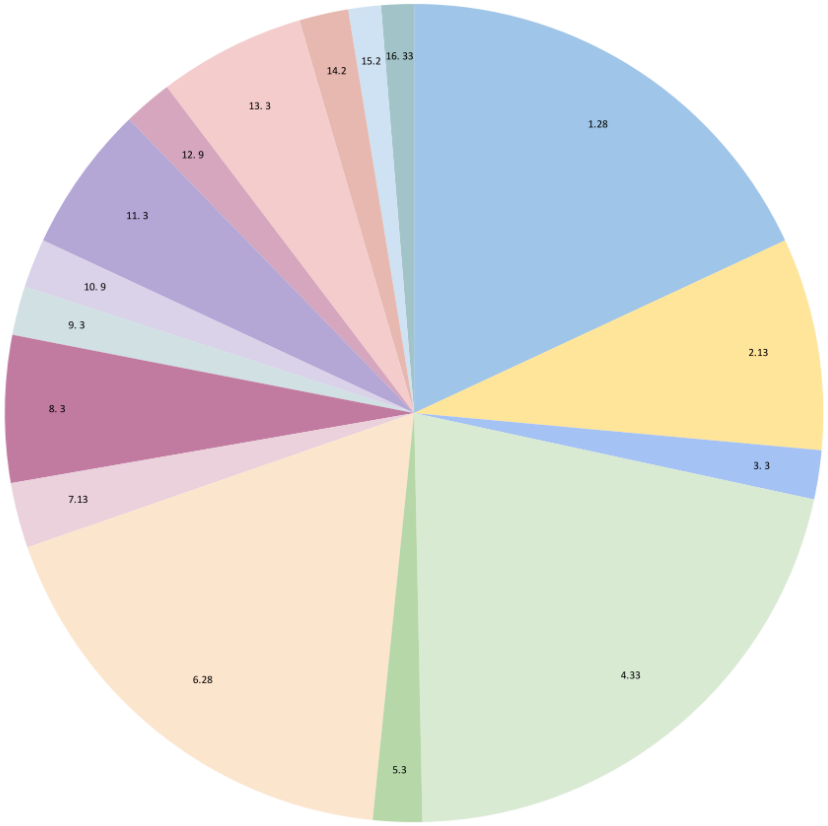
φ **U** true = true

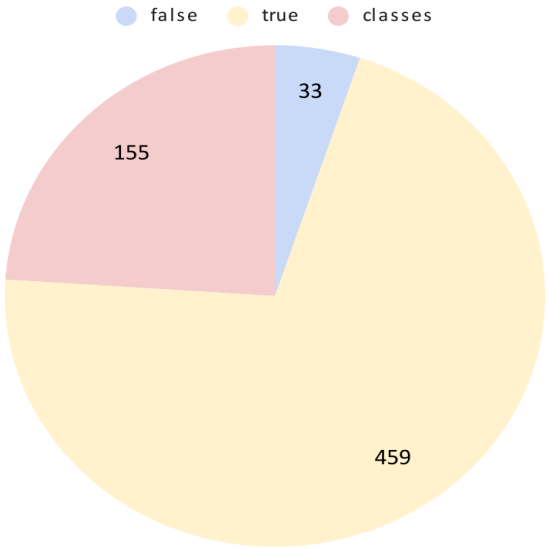
φ **U** φ = φ

$\neg\varphi$ **U** φ = **F** φ

$(\varphi \wedge \chi)$ **U** $(\varphi \vee \psi)$ = $\varphi \vee \psi$

Classification by Simplification: Classes





1. G(rel)
2. G(inv)
3. G(¬fin)
4. G(¬trig)
5. G(inv ∧ ¬fin)
6. G(trig → rel)
7. G(trig → inv)
8. G(trig → G(inv))
9. G(trig → G(¬fin))
10. G(trig → G(inv ∧ ¬fin))
11. G(¬rel → (inv ∧ (G(inv) ∨ (inv U rel))))
12. G(¬rel → (G(¬fin) ∨ (¬fin U (rel ∨ fin))))
13. G((trig ∧ ¬rel) → (inv ∧ (G(inv) ∨ (inv U rel))))
14. G((trig ∧ ¬rel) → (G(¬fin) ∨ (¬fin U (rel ∨ fin))))
15. G(inv ∧ (G(inv ∧ ¬fin) ∨ ((inv ∧ ¬fin) U (fin ∧ inv))))
16. G(trig → (inv ∧ (G(inv ∧ ¬fin) ∨ ((inv ∧ ¬fin) U (fin ∧ inv)))))

Classification by Simplification: Classes

№	Class formula	capacity
1	G(rel)	28
2	G(inv)	13
3	G(¬fin)	3
4	G(¬trig)	33
5	G(inv ∧ ¬fin)	3
6	G(trig → rel)	28
7	G(trig → inv)	4
8	G(trig → G(inv))	9
9	G(trig → G(¬fin))	3
10	G(trig → G(inv ∧ ¬fin))	3
11	G(¬rel → (inv ∧ (G(inv) ∨ (inv U rel))))	9
12	G(¬rel → (G(¬fin) ∨ (¬fin U (rel ∨ fin))))	3
13	G((trig ∧ ¬rel) → (inv ∧ (G(inv) ∨ (inv U rel))))	9
14	G((trig ∧ ¬rel) → (G(¬fin) ∨ (¬fin U (rel ∨ fin))))	3
19	G(inv ∧ (G(inv ∧ ¬fin) ∨ ((inv ∧ ¬fin) U (fin ∧ inv))))	2
16	G(trig → (inv ∧ (G(inv ∧ ¬fin) ∨ ((inv ∧ ¬fin) U (fin ∧ inv)))))	2
19	false	33
18	true	459

Simplification and Classification: Applications

Converting EDTL requirements into LTL formulas

- model checkers
- generating test scenarios
- the classification of EDTL requirements

The classification of EDTL requirements

- diagnosing errors when specifying requirements
- developing a canonical form of requirements,
- defining attribute default values
- developing methods for explaining EDTL requirements in natural language

NL-translation of EDTL Requirements

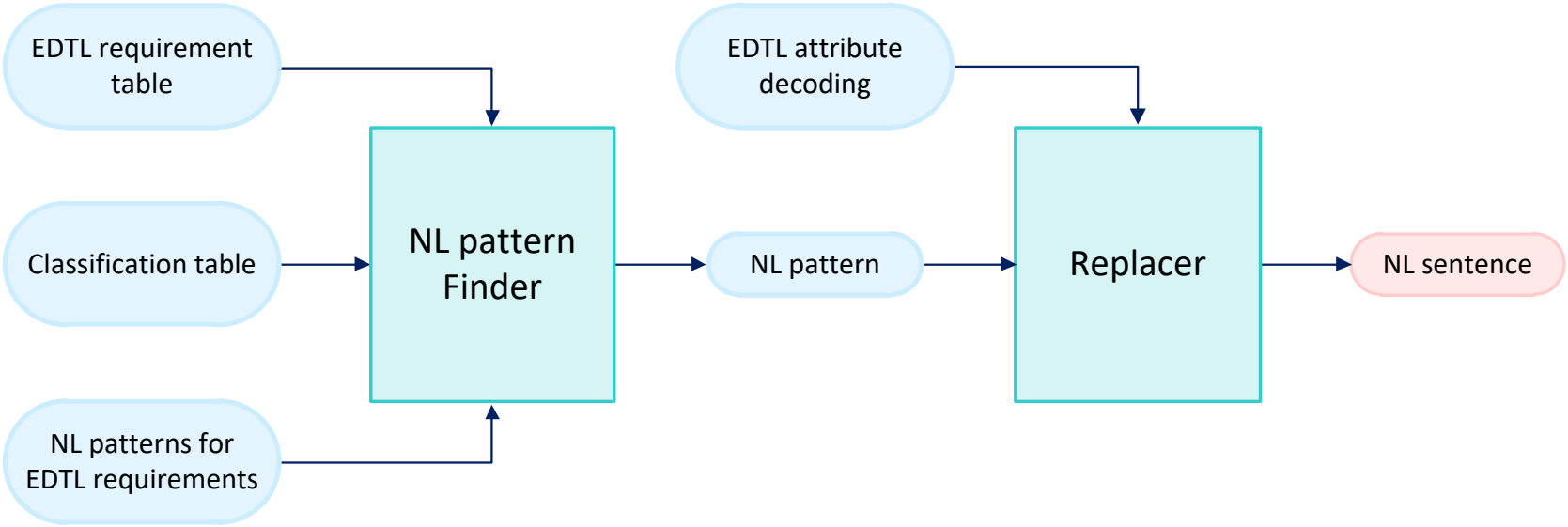
NL-translation: Motivation

- If the dryer (D) was not turned on and hands (H) appeared, it will turn on ASAP

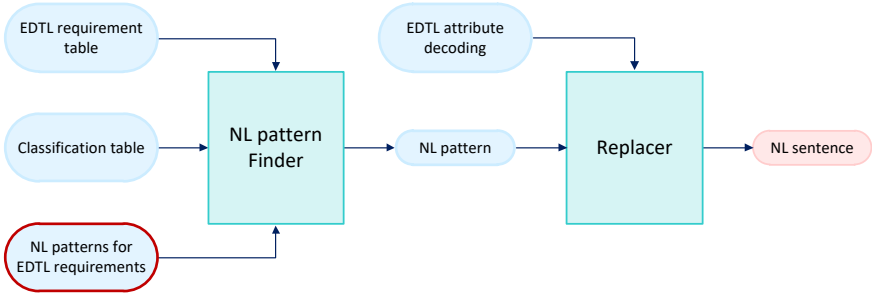
trigger	release	final	delay	invariant	reaction
/H ∧ ¬D	*false*	*true*	*true*	*true*	D’

- **G**(trig → rea)
- **G**(/H ∧ ¬D → D’)

NL-translation: the Algorithm



NL-translation: ToDo

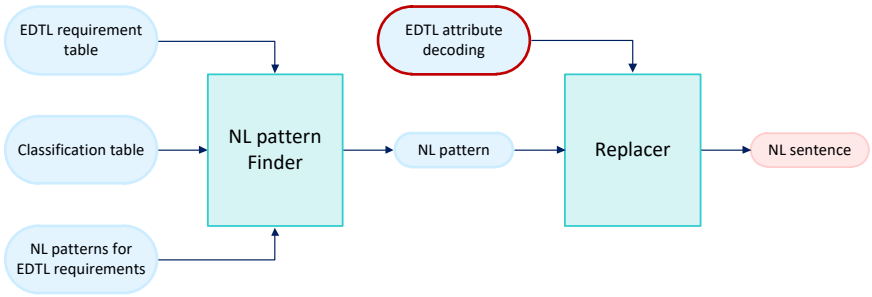


- NL patterns for EDTL requirements
  - EDTL representation

trigger	release	final	delay	invariant	reaction
trig	*false*	*true*	del	*true*	rea

    - The reaction <rea> to the <trig> event must appear no later than the <del> event
  - LTL representation
    - **G** (inv)
      - Always <inv>
  - Corpus of requirements
    - If there are no further requests, the elevator must stop and become idle.
    - The doors must always be closed when the elevator is moving.
    - When <trig> then <inv>.

NL-translation: ToDo



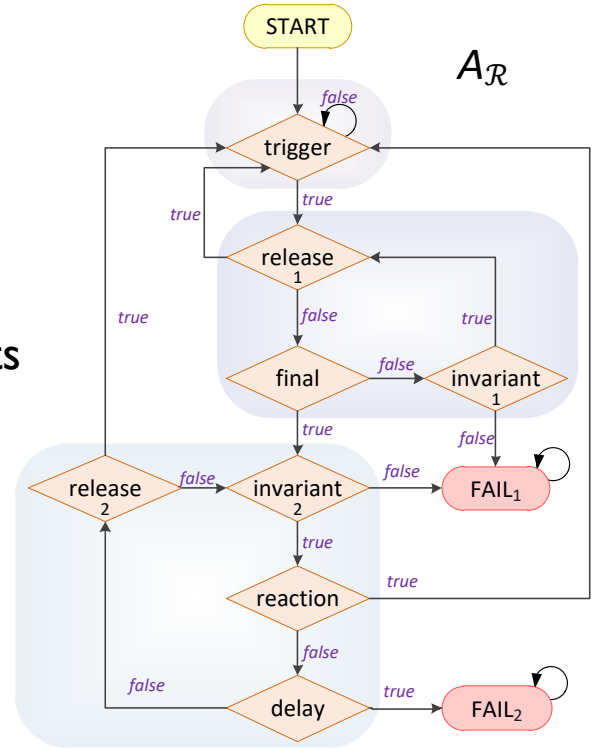
- EDTL attribute decoding
  - The reaction <rea> to the event <trig> must appear no later than the <del> event
    - trig = /RcvdX – signal X is received
    - del = passed(2s) – 2s passed
    - rea = X_rcvdBit – set bit *X_received*
    - The reaction "set bit *X_received*" to the event "signal X is received" must appear no later than the "2s passed" event
    - When signal X is received, the system shall set the signal X received bit within 2 seconds

    - trig = /open – just opened
    - del = passed(10s) – 10s passed
    - rea = ¬open – closed
    - The reaction "closed" to the event "just opened" must appear no later than the "10s passed" event
    - The open signal must be true for no more than 10 seconds.

Model-Checking EDTL Requirements

Model-Checking EDTL: the Idea

- Reduce CS state space w.r.t. input and output points
- Construct Buchi automaton in advance



$\Phi_{\mathcal{R}}$ =

G(trigger ∧ ¬release → invariant ∧ (**G**(invariant ∧ ¬final) ∨
(invariant ∧ ¬final **U** release ∨ (final ∧
(invariant ∧ ¬delay **U** (release ∨ (invariant ∧ reaction)))))))).

EDTL Requirements: ToDo

1. Automata semantics – prove equivalence with FOL, LTL
2. FOL semantics – prove equivalence with Aut, LTL
3. LTL semantics – prove equivalence with FOL, Aut
4. Bounded checking algorithm
5. Consistency checking – implement
6. Automatic translation to LTL – add Boolean translations
7. Semantic classification – refinement rules
8. NL representation – improve
9. Automatic generation of dynamic verifier
10. The corpus of EDTL requirements – extend
11. Model checking (the idea) – refine and develop

Plans

- Develop *consistency-checking* methods for EDTL requirements
- Formally prove
  - the *equivalence* of LTL and FOL semantics
  - *soundness* of the bounded-checking algorithm
- Develop and implement *EDTL-specialized verification*
  - dynamic verification
  - model checking
  - deductive verification
- Add support for *pattern composition* to the notation