

Industrial Verification of Avionic, Automotive, and Railway Systems – Practical Applications and Theoretical Foundations –

Jan Peleska

University of Bremen and Verified Systems International GmbH

jp@cs.uni-bremen.de

Background

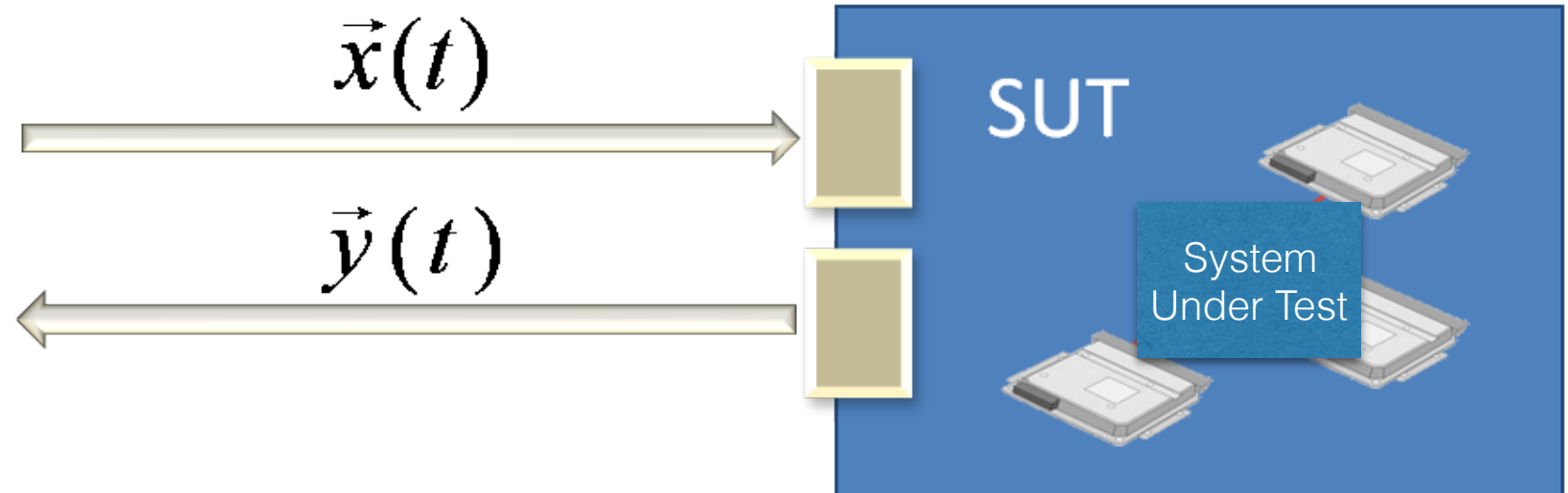
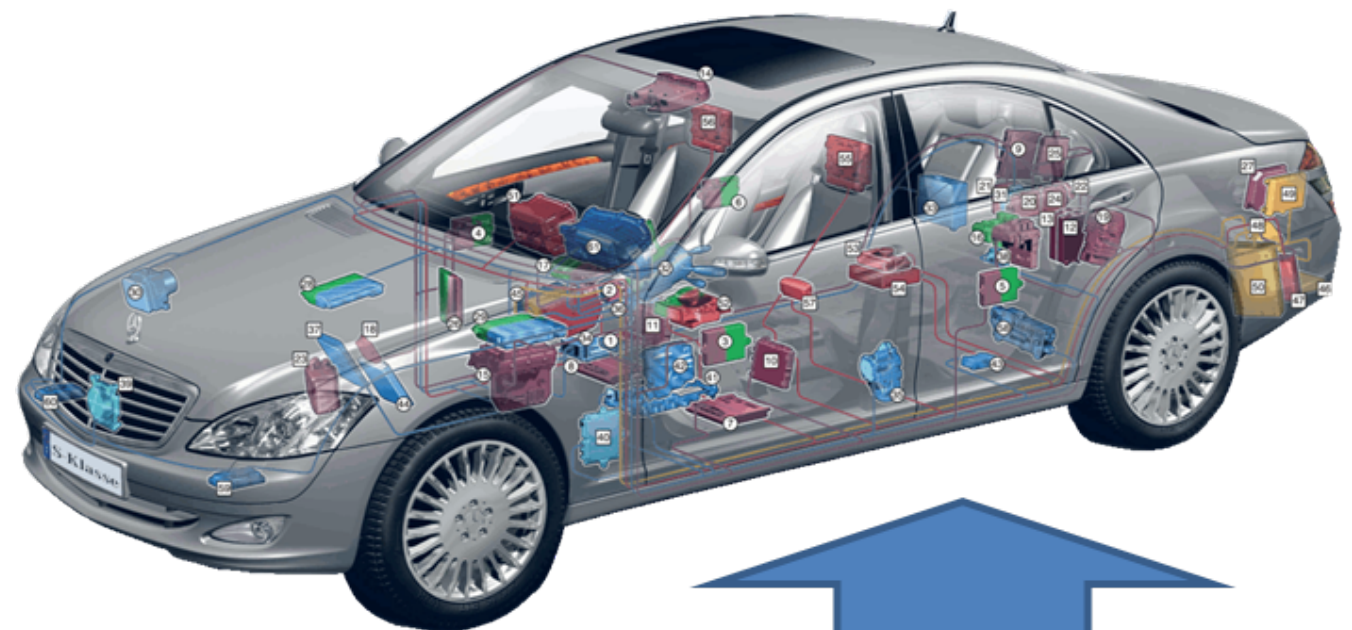
- **My research group at the University of Bremen**
 - Specialised on verification of safety-critical control systems
 - Focus on test automation
 - Collaboration with Prof. Dr. habil. Wen-ling Huang in this field

Background

- **Verified Systems International GmbH**
 - Founded 1998 as a spinoff company from the University of Bremen
 - Specialised on verification and validation of safety-critical systems – aerospace, railways, automotive
 - Tool development, hardware-in-the-loop test bench development, and service provision
 - Main customers Airbus, Siemens
 - 25 employees

Hardware-in-the-Loop Test Benches

For testing integrated HW/SW systems



An Observation . . .

- Many maths and computer science students in their first semesters seem to think that complex theory and difficult algorithms are only needed to pass exams . . .
- This is not true!
- Many of the most important innovations and products are based on highly complex mathematical foundations and on sophisticated software algorithms
- I will try to illustrate this during my talk

“Program testing can best show the presence of errors but never their absence.”

Edsger W. Dijkstra, 1930 – 2002

“Program testing can best show the presence of errors but never their absence.”

Edsger W. Dijkstra, 1930 – 2002

This famous quotation was based on a number of well-observed facts

- Control systems that are supposed to run 24/7 – that is, without termination – may perform an uncountable number of different input/output sequences of infinite length
- For systems with state components of infinite type (time, speed, temperature,...) it is impossible to enumerate all possible state values during testing

“Program testing can best show the presence of errors but never their absence.”

Edsgar W. Dijkstra, 1930 – 2002

This famous quotation was based on a number of well-observed facts

- Control flow that is uncoupled from sequence

Sorry, Edsgar, but this statement is actually not true in this generality!

- For systems with infinite state vectors (time, speed, temperature,...) it is impossible to enumerate all possible state values during testing

“Program testing can best show the presence
of errors but never their absence.”

Not true in this generality

Edsger Dijkstra – 2002

Since about 1975, many computer scientists and mathematicians have contributed results implying that

“Under certain hypotheses, you can **prove** or **disprove** the correctness of “infinite” systems by a **finite number of tests.**”

“Program testing can best show the presence
of errors but never their absence.”

Edsger Dijkstra – 2002

Not true in this generality

Since about 1975, many computer scientists and mathematicians have contributed results implying that

“Under certain hypotheses, you can
disprove the correctness of “infinitely many”
a **finite number of tests**.”

. . . . and this can be fully automated

Objectives

- Explain how **safety-critical embedded systems** are verified
- Illustrate how **complex mathematics and computer science techniques are needed** to
 - Invent trustworthy verification methods
 - Create tools that implement these methods in the most efficient way

Objectives

- Show that **automation is necessary**:
 - otherwise these effective, but highly complex methods could never be applied in practice
- Present some **novel research results with significant practical implications**

Overview

1. Safety-critical systems – examples
2. Verification requirements of international standards
3. Theory meets innovation
 - A. Complete theories for testing with guaranteed fault coverage
 - B. Theory translation for testing: a new complete equivalence class testing strategy
 - C. Theory translation for property checking: a complete testing strategy for checking safety properties

Safety-critical systems

- Examples

Safety-critical Systems

A . . . **safety-critical system** is a system whose failure or malfunction may result in one (or more) of the following outcomes:

- death or serious injury to people
- loss or severe damage to equipment/property
- environmental harm

Safety-critical Systems – Examples

Airbag controller



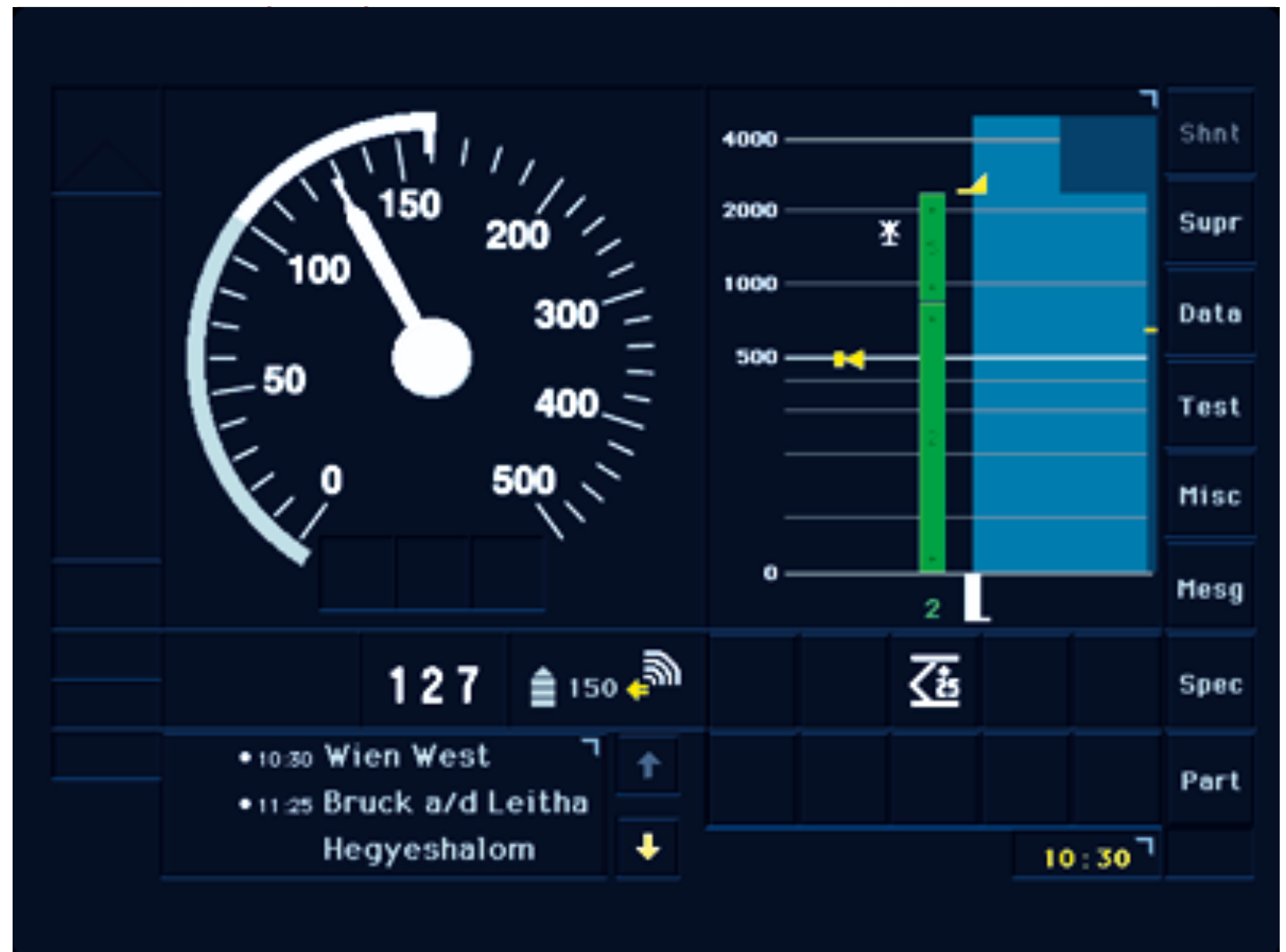
Safety-critical Systems – Examples

Aircraft thrust
reversal



Safety-critical Systems – Examples

Train speed
supervision



Verification Requirements of International Standards

Verification Requirements of International Standards

- Safety-critical systems development and verification is controlled by laws
- These laws state that development and verification must follow the rules specified in applicable standards
- For developing control systems, the following standards apply
 - Avionic domain: **RTCA DO-178C**
 - Railway domain: **CENELEC EN50128:2011**
 - Automotive domain: **ISO 26262**

Verification Requirements of International Standards

- These standards differ in many details, but they contain some basic requirements
 - Development must be based on **requirements**
 - **Structural requirements**: interfaces, system components
 - **Behavioural (functional) requirements**: how inputs are transformed into outputs
 - Every piece of software code must be **traced** back to at least one requirement

- For the most critical applications, requirements should be expressed by **formal models** with mathematical interpretation
 - **Syntax and static model semantics**: is the model well-formed?
 - **Behavioural model semantics**: how does the model state, including inputs and outputs, evolve over time?
- **Requirements (models) must be verified**
 - Internally consistent?
 - Consistent with related more abstract requirements
 - For example, a software requirement may be traced back to a system requirement

- **Code must be verified**
 - Does it implement the related requirements correctly?
 - Verification is preferably performed by **testing** the software integrated in the controller's hardware
- Verification results need to be checked with respect to completeness and correctness
- Tools automating development or verification steps need to be **qualified**

Theory meets Innovation –
A. Complete Theories for Testing
With Guaranteed Fault Coverage

Testing With Guaranteed Fault Coverage

- **Objective**
 - Can we construct **complete** test suites that guarantee to uncover every implementation fault ?
 - Can we derive these test suites from models of the control system's required behaviour?
- **Answer**
 - Yes, under certain hypotheses expressed by **fault models**

Complete Test Suites

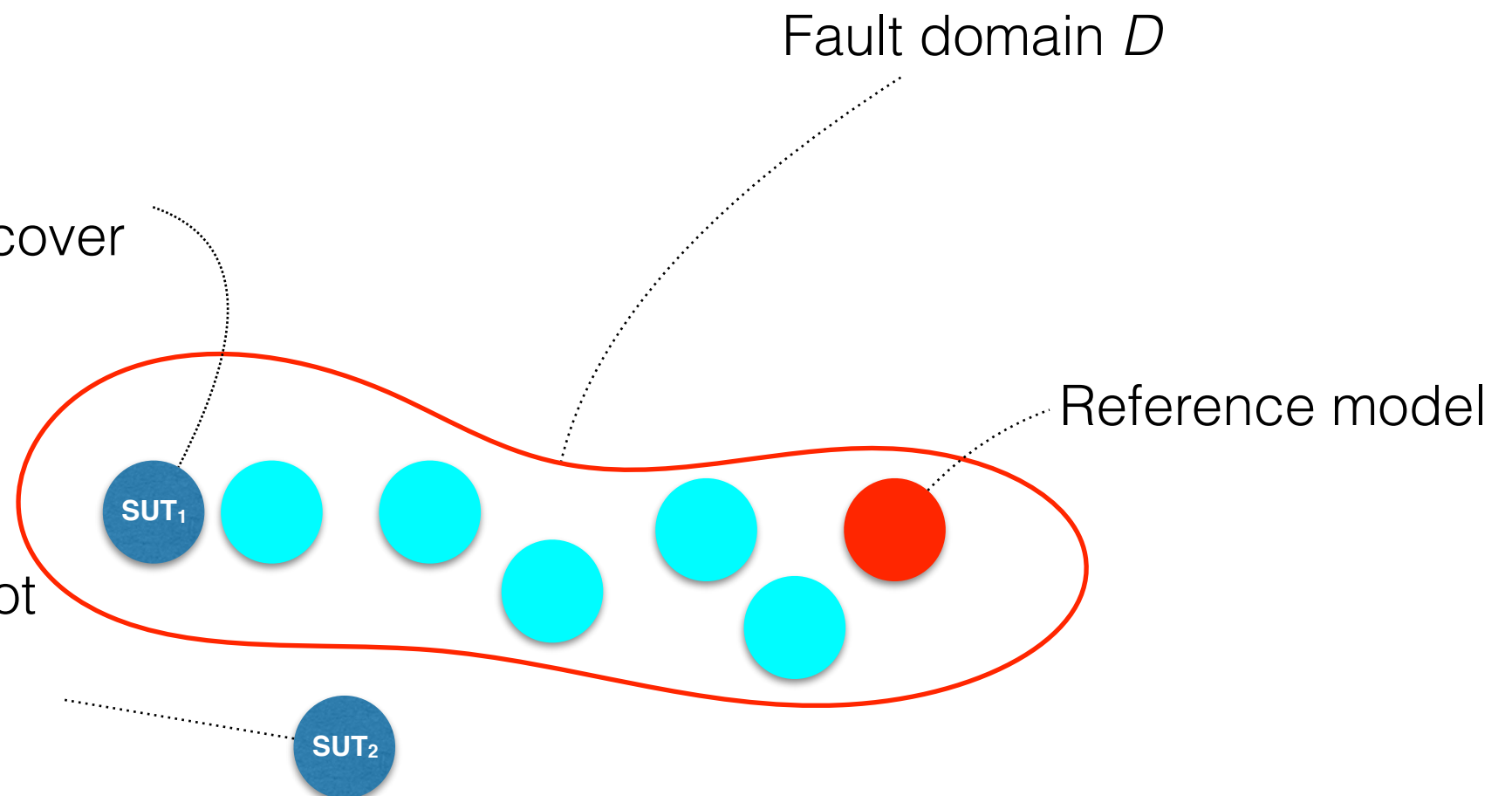
- Defined with respect to a **fault model** consisting of
 - a reference model M
 - a conformance relation \cong for comparing other models to M
 - a fault domain Dom
- **Complete** = sound + exhaustive
- **Sound** = every M' in Dom satisfying $M' \cong M$ passes
- **Exhaustive** = every M' in Dom violating $M' \cong M$ fails

Complete Test Suites

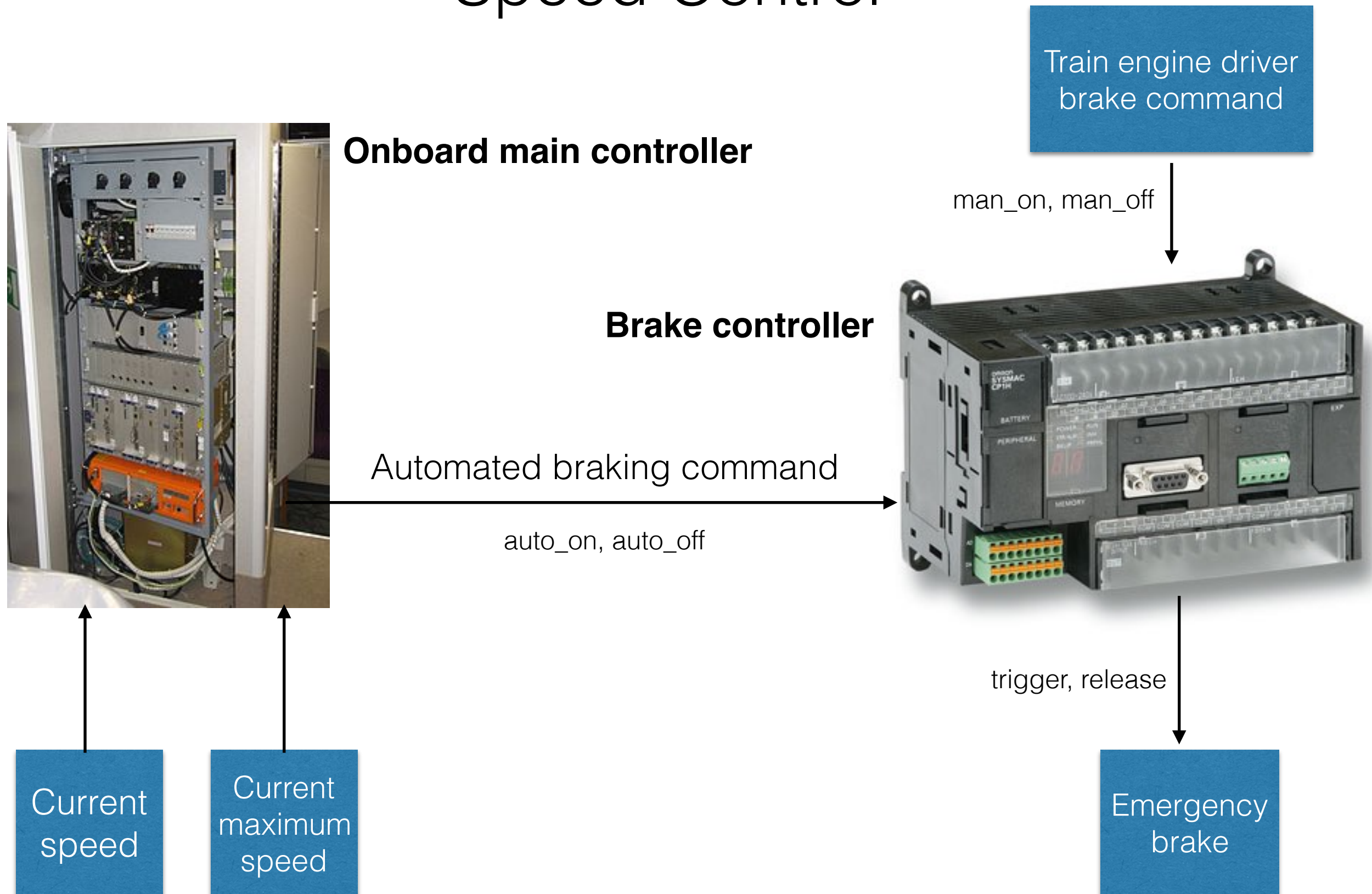
- For black-box testing, completeness depends on a pre-specified **fault domain**
- The true behaviour of the system under test must be captured in a (very large) class of models that may or may not be correct in relation to the given reference model

True behaviour of SUT₁ –
complete test suite for D will uncover
every deviation from reference
model

True behaviour of SUT₂ –
complete test suite for D may not
uncover every deviation
from reference model



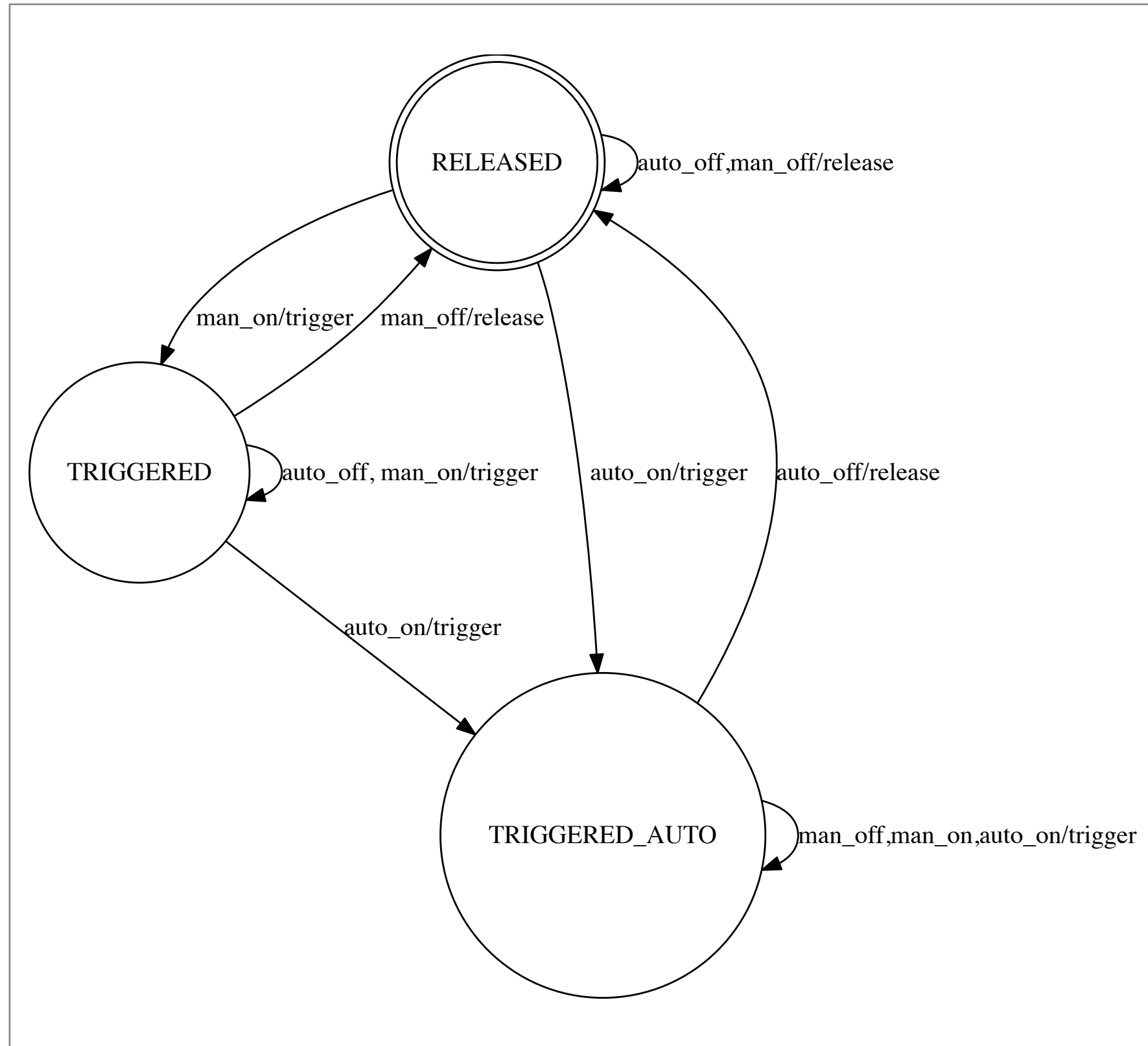
Application Scenario – Train Onboard Speed Control



Mathematical model:
Finite State Machine

- Discrete input events
- Discrete internal state
- Discrete output events

Brake Controller

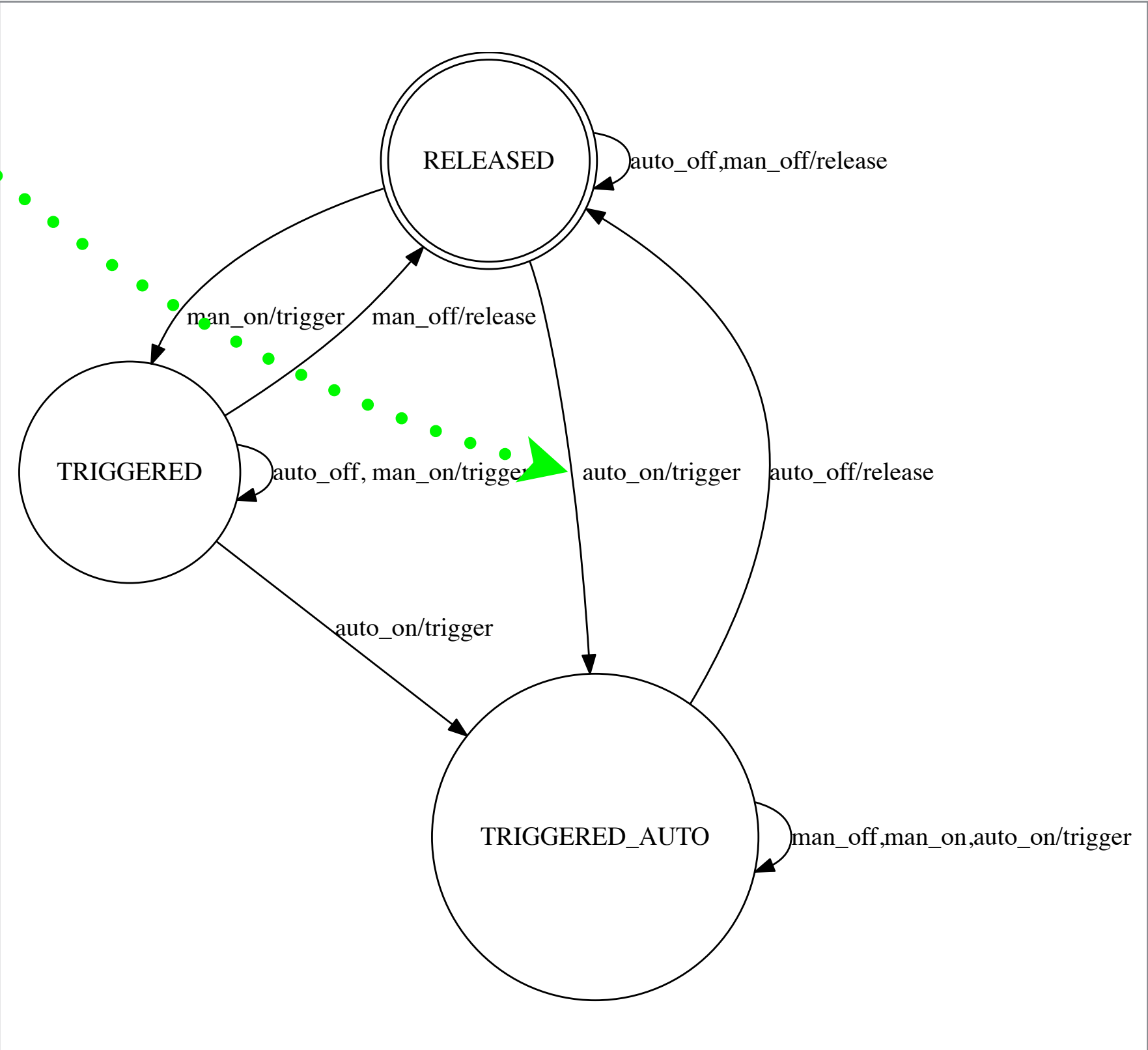


Finite State Machines – Behavioural Semantics

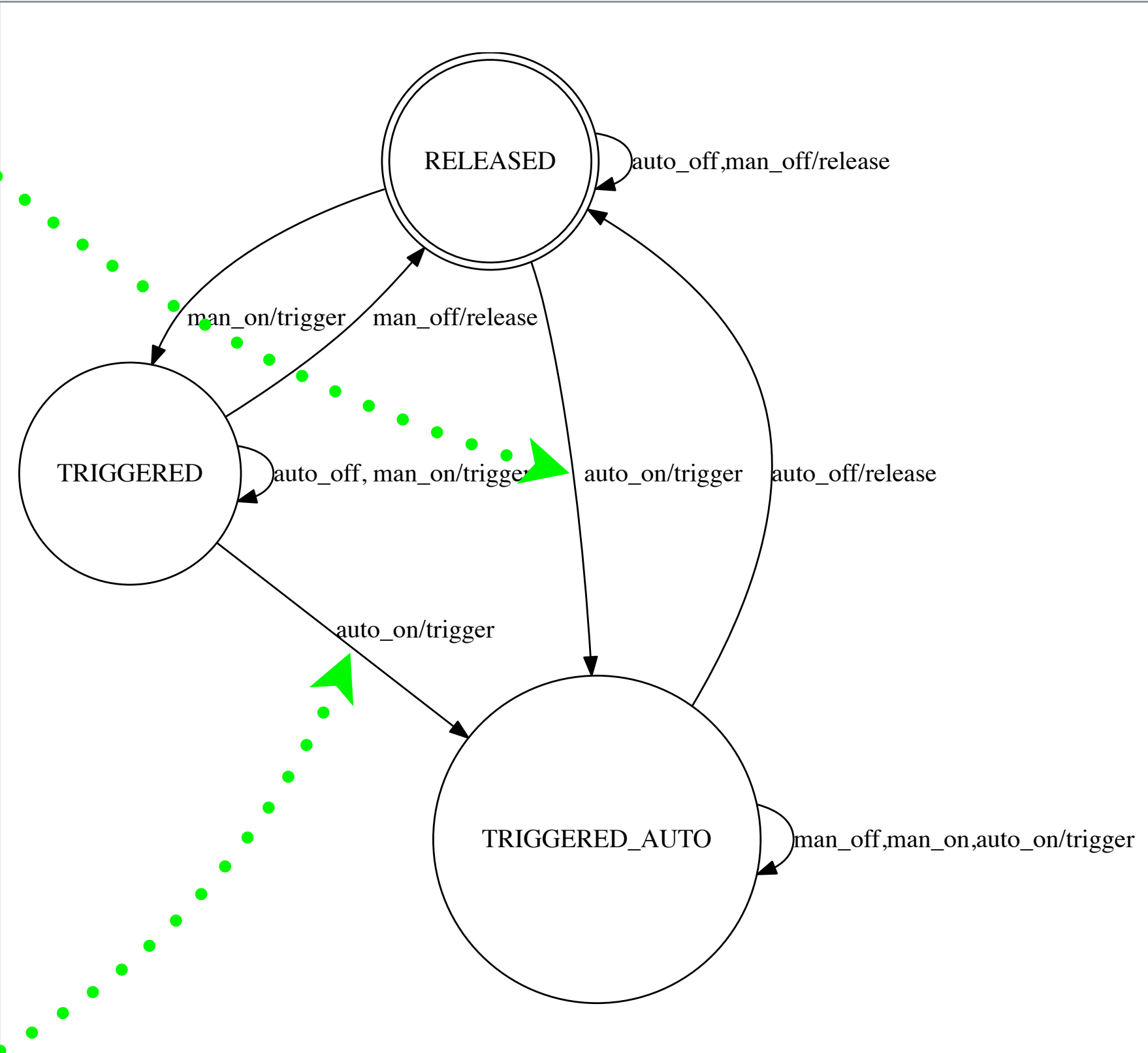
$$M = (Q, q_0, I, O, h)$$

- $Q \neq \emptyset$: finite set of states
- $q_0 \in Q$: initial state
- $I \neq \emptyset$: finite set of input alphabet
- $O \neq \emptyset$: finite set of output alphabet
- $h \subseteq Q \times I \times O \times Q$: transition relation

$(RELEASED, auto_on, trigger, TRIGGERED_AUTO) \in h$



$(\text{RELEASED}, \text{auto_on}, \text{trigger}, \text{TRIGGERED_AUTO}) \in h$



$(\text{TRIGGERED}, \text{auto_on}, \text{trigger}, \text{TRIGGERED_AUTO}) \in h$

Example of a complete Test Method: the W-Method

M. P. Vasilevskii 1973 and Tsun S. Chow 1978

- The W-Test Suite

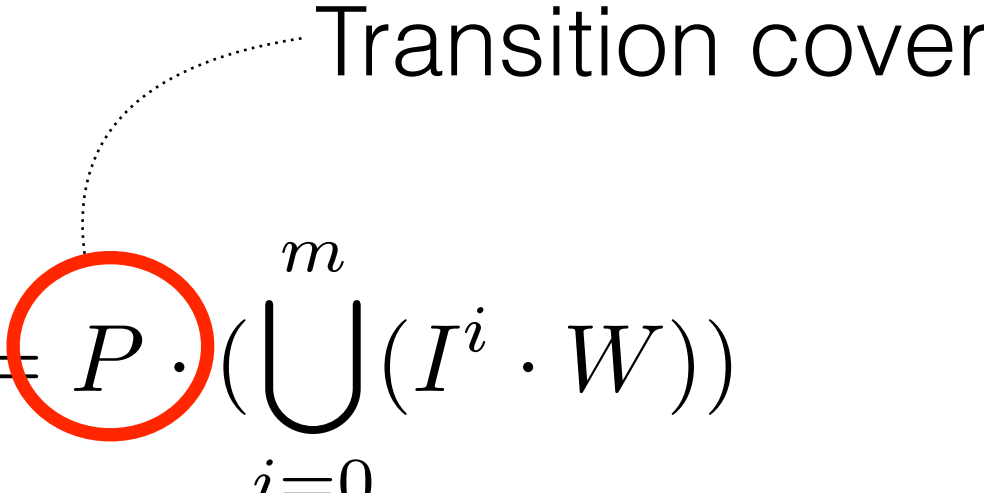
$$\mathcal{W}(A) = P \cdot \left(\bigcup_{i=0}^m (I^i \cdot W) \right)$$

$P \cdot Q$ is defined for sets of traces.

$$P \cdot Q = \{p.q \mid p \in P \wedge q \in Q\}$$

Example for a complete Test Method: the W-Method

Transition cover

$$\mathcal{W}(A) = P \cdot \left(\bigcup_{i=0}^m (I^i \cdot W) \right)$$


$P \cdot Q$ is defined for sets of traces.

$$P \cdot Q = \{p.q \mid p \in P \wedge q \in Q\}$$

Example for a complete Test Method: the W-Method

Input traces of length i

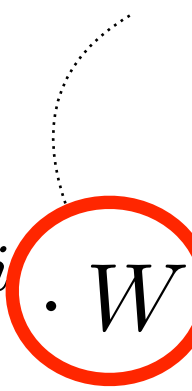
$$\mathcal{W}(A) = P \cdot \left(\bigcup_{i=0}^m (I^i \cdot W) \right)$$

$P \cdot Q$ is defined for sets of traces.

$$P \cdot Q = \{p.q \mid p \in P \wedge q \in Q\}$$

Example for a complete Test Method: the W-Method

Characterisation set

$$\mathcal{W}(A) = P \cdot \left(\bigcup_{i=0}^m (I^i \cdot W) \right)$$


$P \cdot Q$ is defined for sets of traces.

$$P \cdot Q = \{p.q \mid p \in P \wedge q \in Q\}$$

Example of a complete Test Method: the W-Method

M. P. Vasilevskii 1973 and Tsun S. Chow 1978

- The W-Test Suite

$$\mathcal{W}(A) = P \cdot \left(\bigcup_{i=0}^m (I^i \cdot W) \right)$$

$P \cdot Q$ is defined for

Theorem. If the implementation

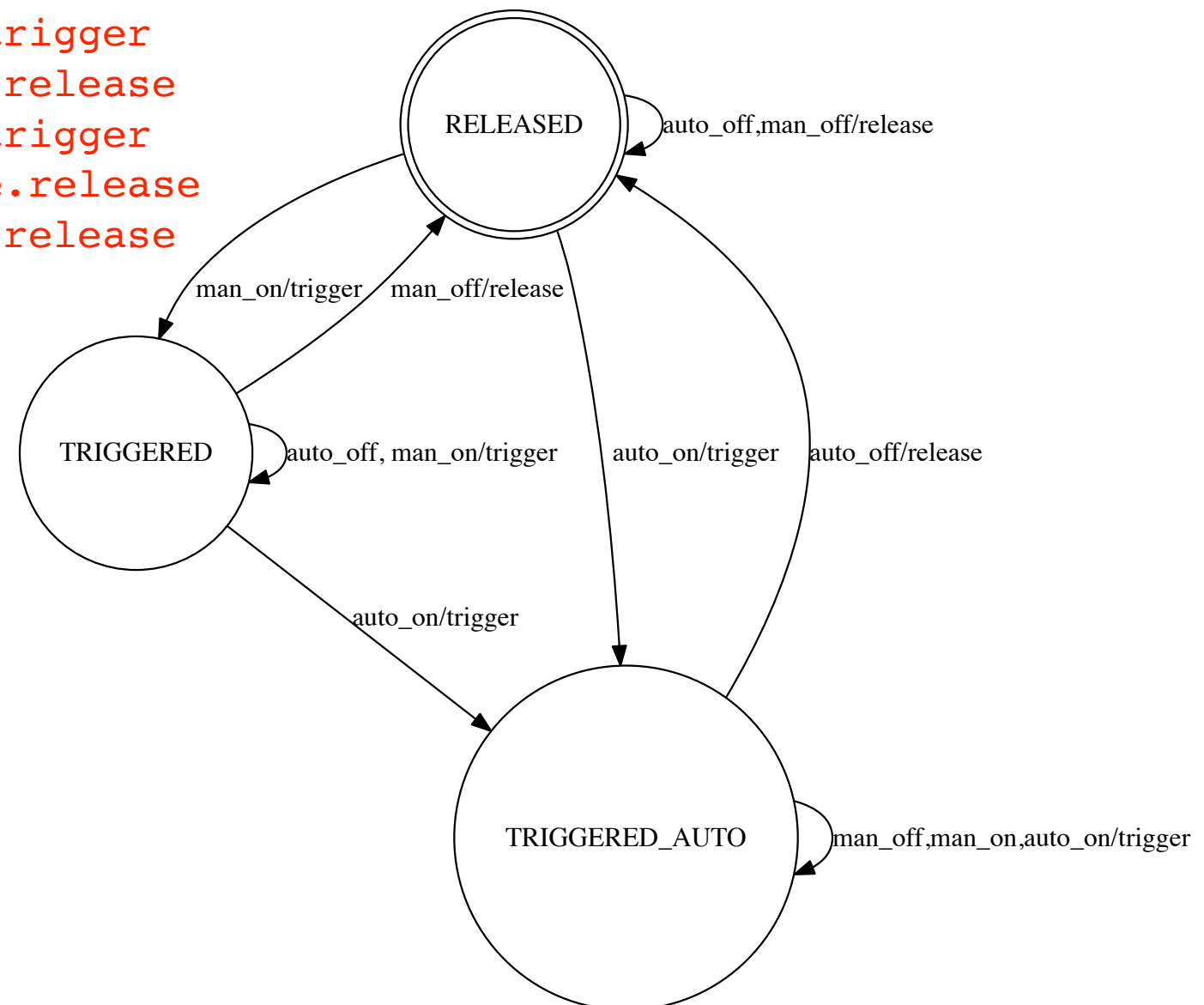
1. Passes all tests specified in $\mathcal{W}(A)$
2. Behaves like an FSM with at most m internal states

then the implementation is I/O-equivalent to the reference FSM: both perform exactly the same input/output sequences

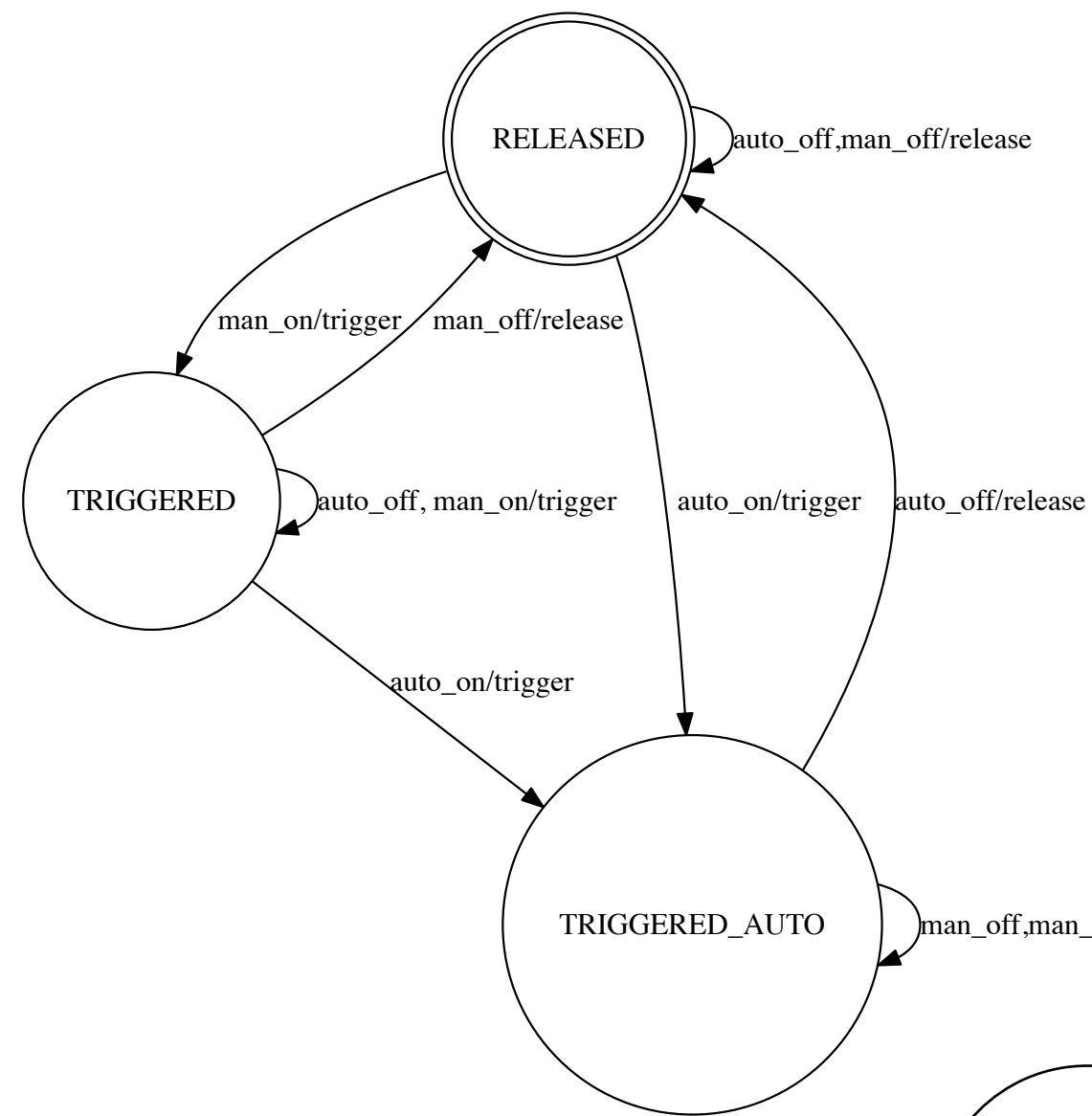
Test cases for hypothesis $m = 3$

This test suite uncovers every error, provided that the implementation has at most 3 states

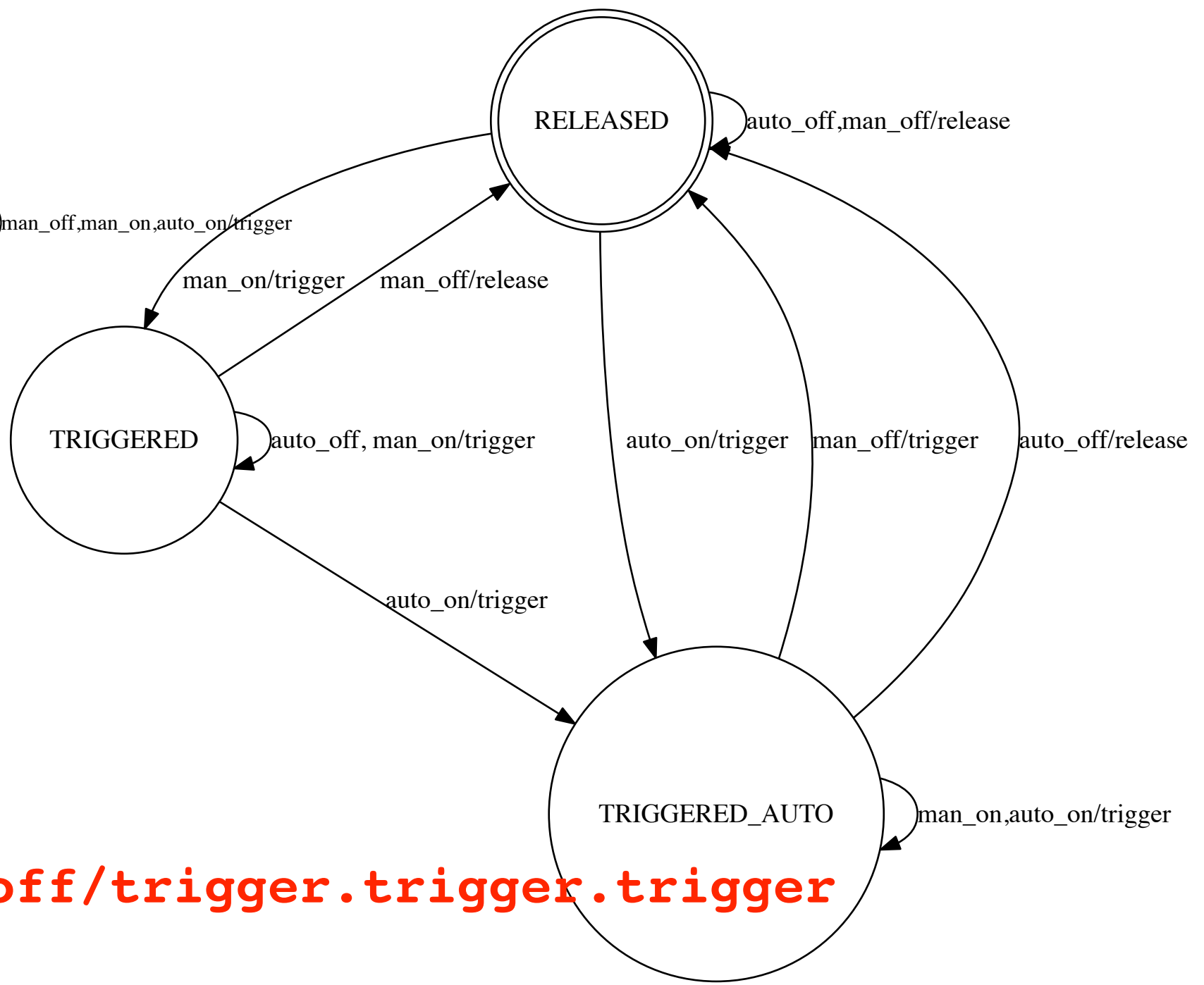
1. man_on.man_on.auto_off/trigger.trigger.trigger
2. man_on.man_on.man_off/trigger.trigger.release
3. man_on.auto_on.auto_off/trigger.trigger.release
4. man_on.auto_on.man_off/trigger.trigger.trigger
5. man_on.man_off.auto_off/trigger.release.release
6. man_on.man_off.man_off/trigger.release.release
7. man_on.auto_off.auto_off/trigger.trigger.trigger
8. man_on.auto_off.man_off/trigger.trigger.release
9. auto_on.man_on.auto_off/trigger.trigger.release
10. auto_on.man_on.man_off/trigger.trigger.trigger
11. auto_on.auto_on.auto_off/trigger.trigger.release
12. auto_on.auto_on.man_off/trigger.trigger.trigger
13. auto_on.man_off.auto_off/trigger.trigger.release
14. auto_on.man_off.man_off/trigger.trigger.trigger
15. auto_on.auto_off.auto_off/trigger.release.release
16. auto_on.auto_off.man_off/trigger.release.release
17. man_off.auto_off/release.release
18. man_off.man_off/release.release
19. auto_off.auto_off/release.release
20. auto_off.man_off/release.release



Reference model



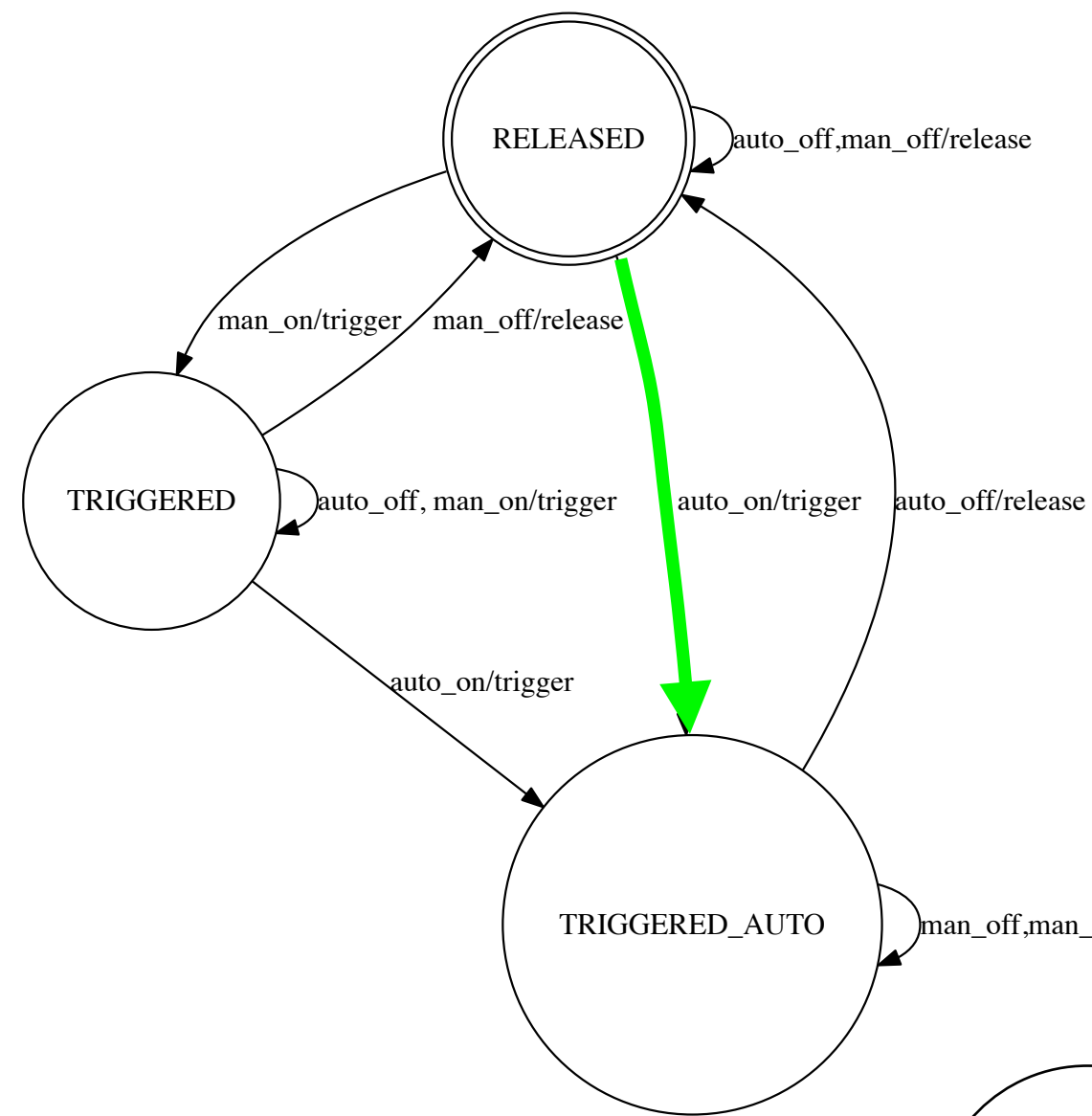
Faulty implementation



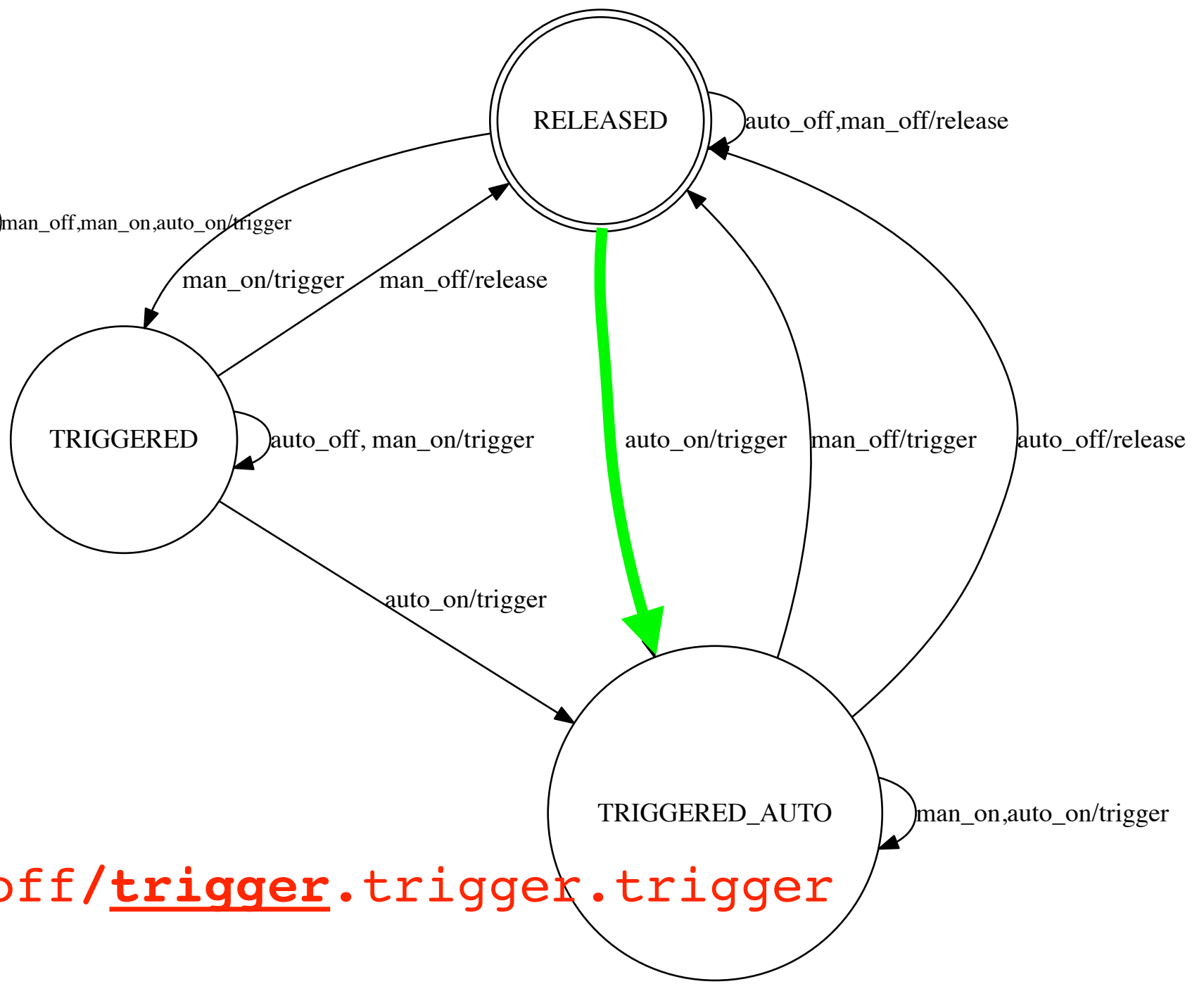
Test case 14.

auto_on.man_off.man_off/trigger.trigger.trigger

Reference model



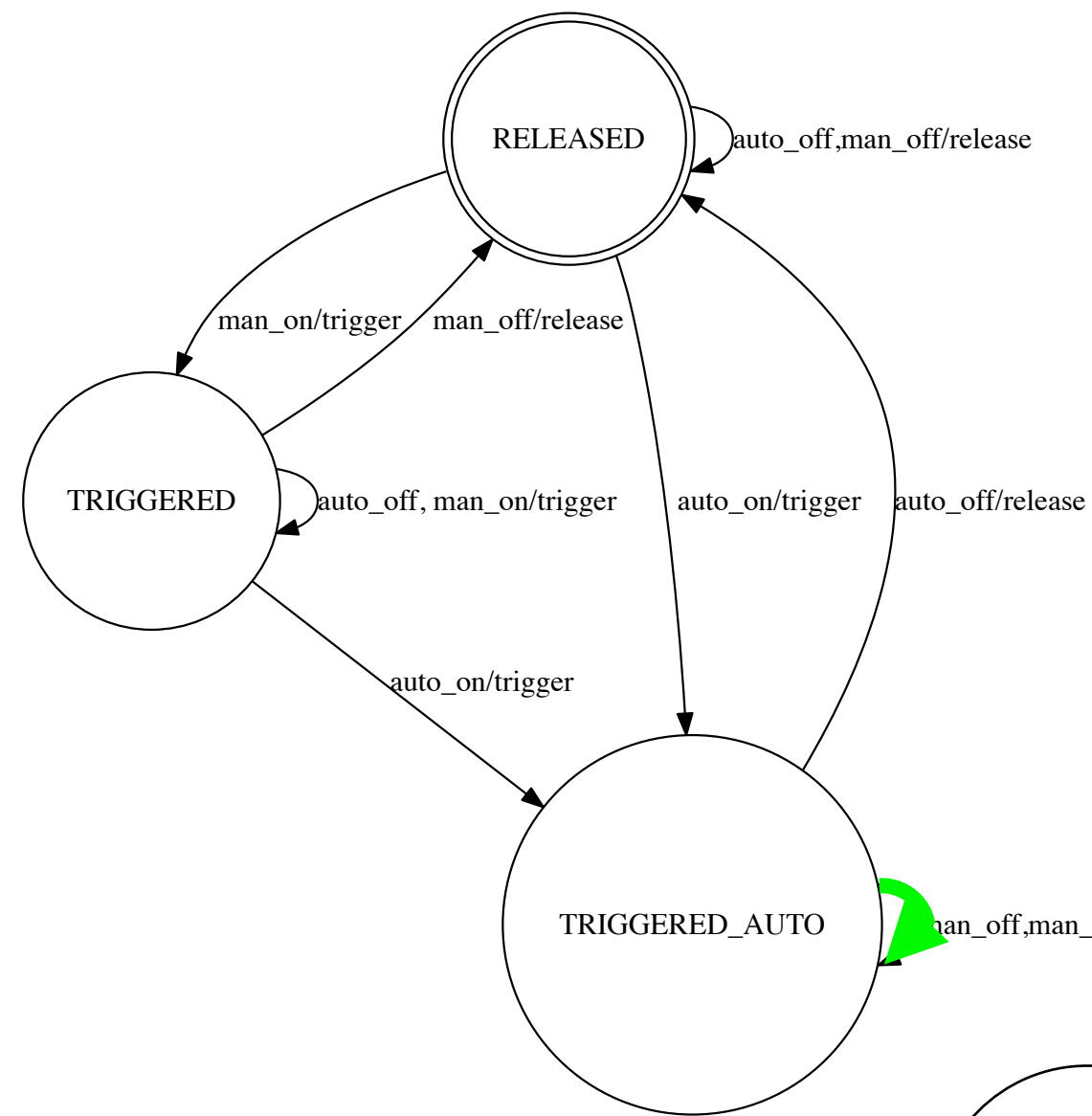
Faulty implementation



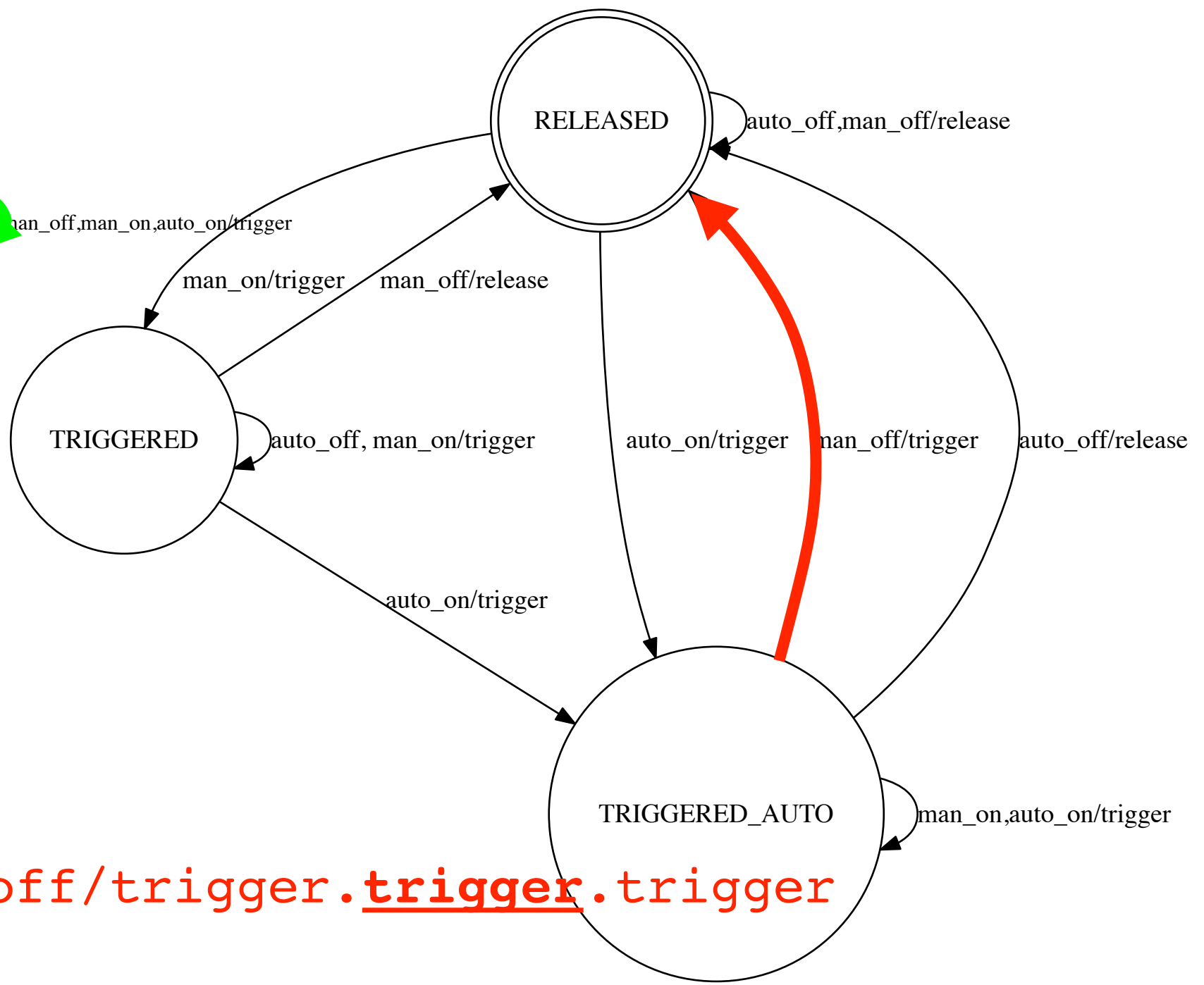
Test case 14.

auto_on.man_off.man_off/trigger.trigger.trigger

Reference model



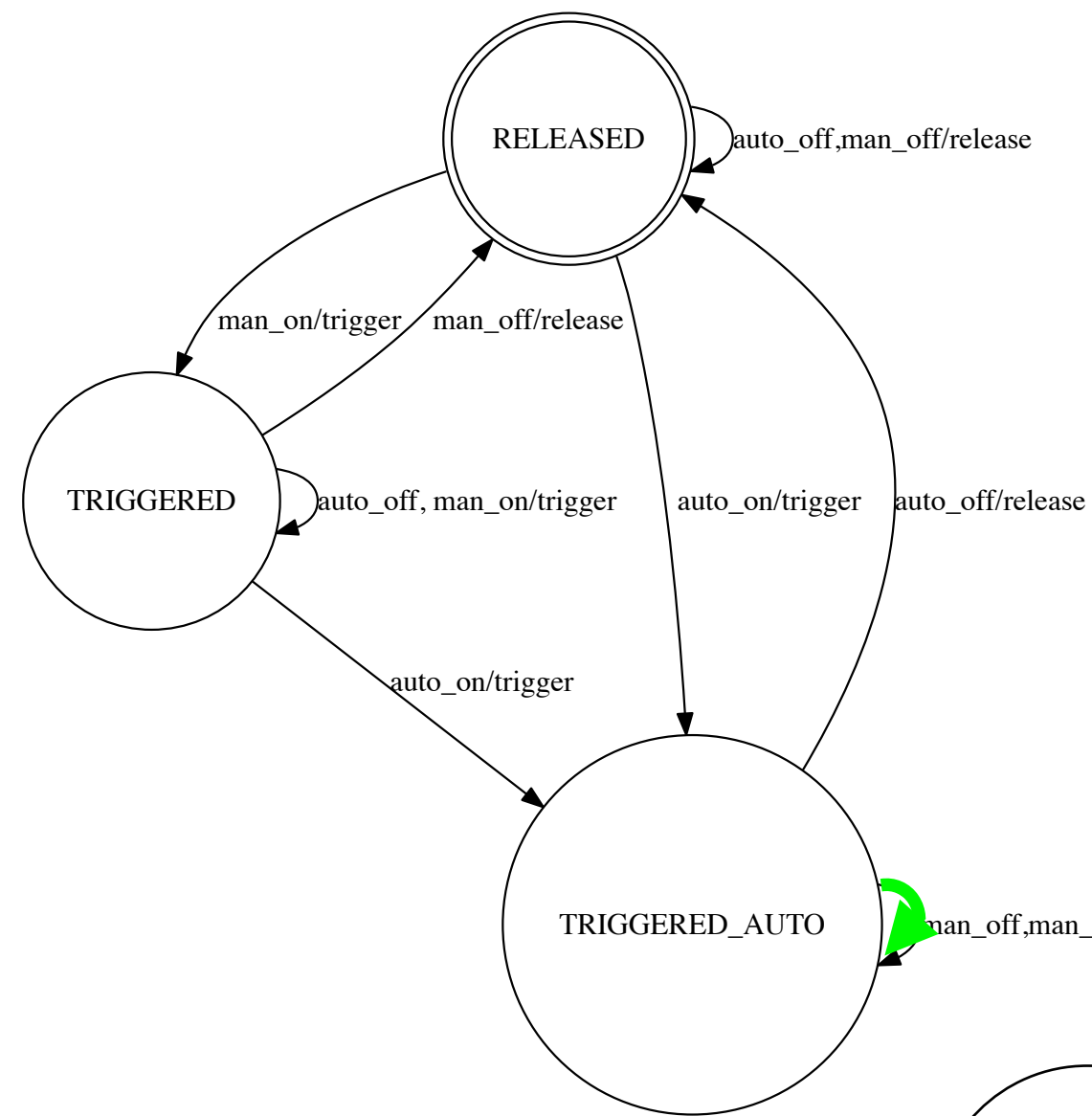
Faulty implementation



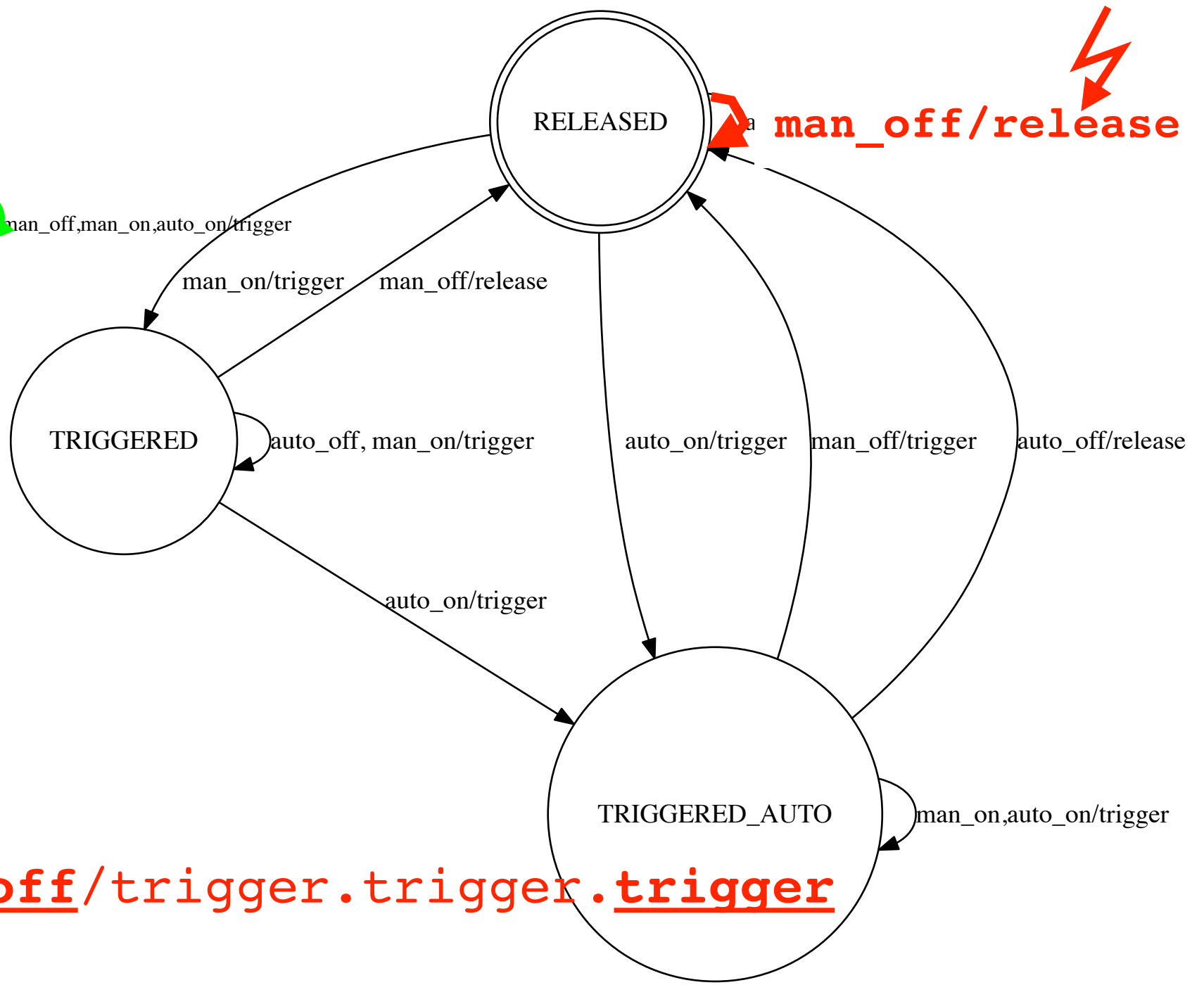
Test case 14.

auto_on.man_off.man_off/trigger.trigger.trigger

Reference model



Faulty implementation



Test case 14.

auto_on.man_off.man_off/trigger.trigger.trigger

Complete Testing Theories

- Since the W-Method described above has been invented, many important extensions have been elaborated
 - **Nondeterministic** finite state machines
 - Testing for **reduction**: system under test implements only a part of the I/O-sequences admissible by the reference FSM

Complete Testing Theories

- Since the W-Method was invented, many important results have been elaborated
- Nondeterministic finite automata
- Testing reduction: system under test implements only a part of the I/O-sequences admissible by the reference FSM

More details about the W-Method are explained in Wen-ling Huang's Alumni Distinguished Lecture Series 'Testing safety-critical discrete state systems – mathematical foundations and concrete algorithms'

Wednesday, 2016-05-25, 10:20 – 12:00, ST527

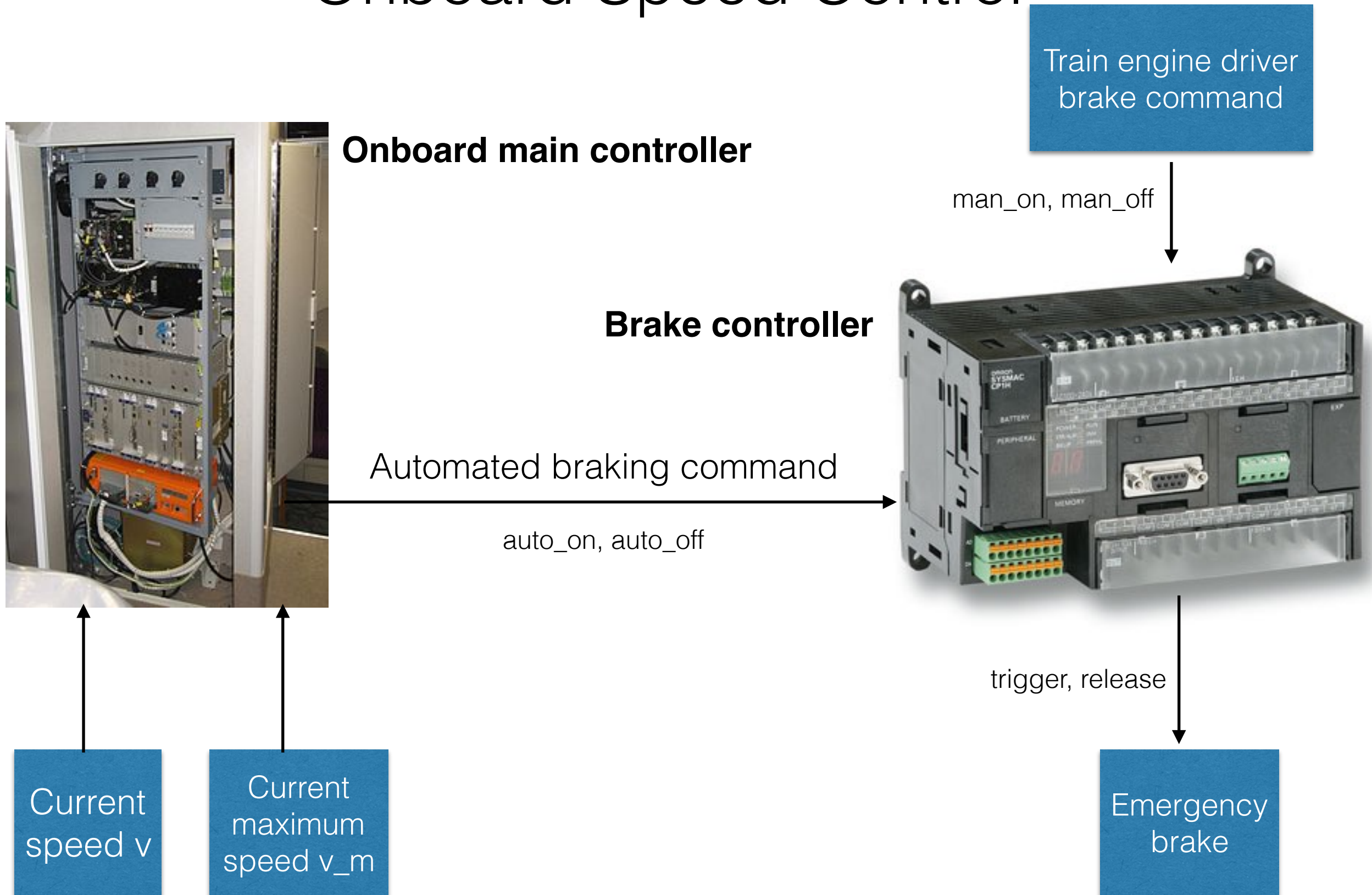
Complete Testing Theories

Which are the benefits from applying complete testing theories?

- **Every error is uncovered** if system under test is inside the fault domain
- Significant **test strength** also for implementations that are outside the fault domain
- Fully justified test cases due to the underlying theory
 - **This facilitates the certification** of a safety critical system: otherwise every test suite has to be justified with respect to its test strength
- Fully **automated** test generation, execution, and evaluation – you just need to construct the reference model

Theory Meets Innovation –
B. Theory translation for testing: a
new complete equivalence class
testing strategy

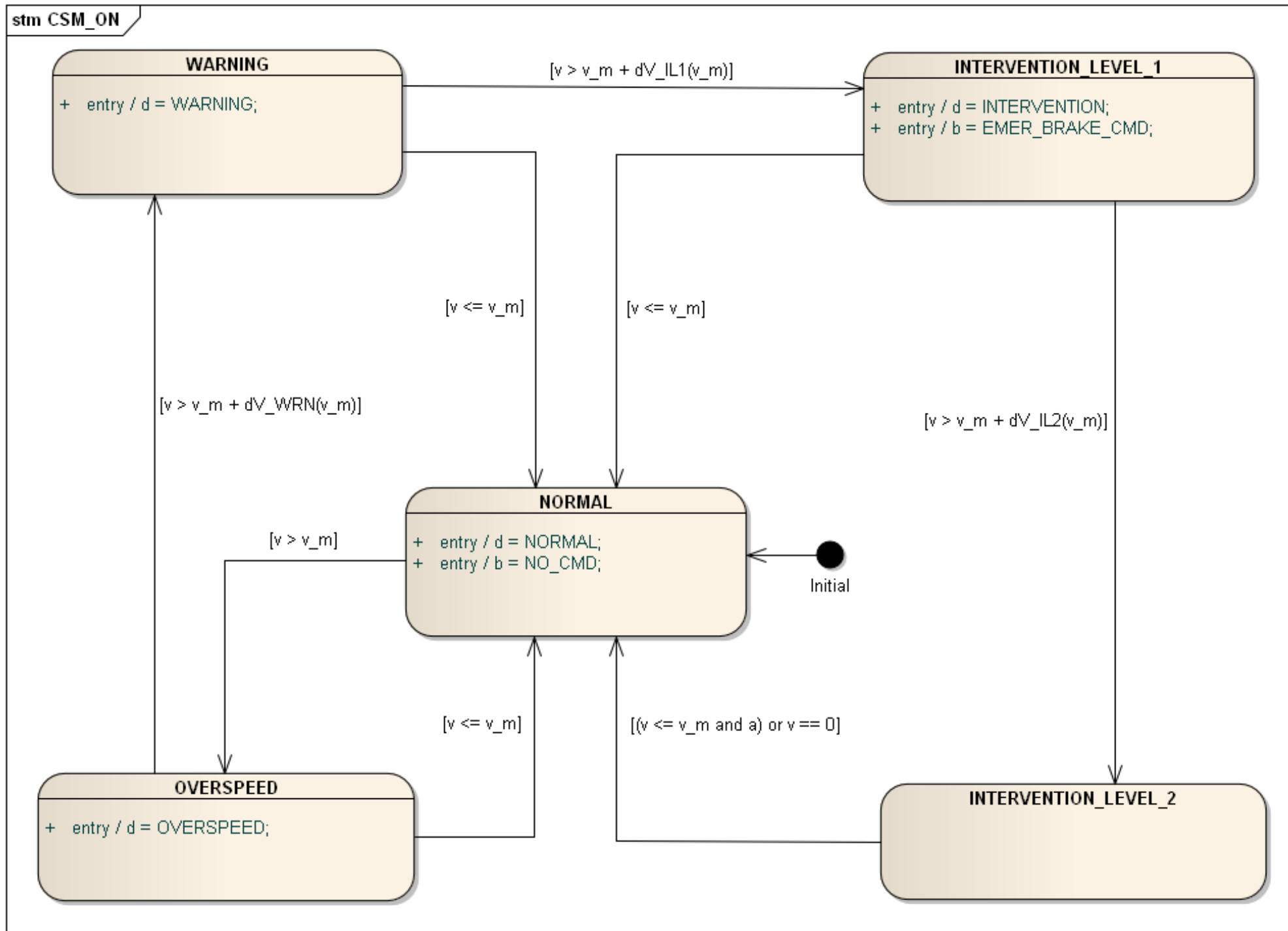
Recall: Application Scenario – Train Onboard Speed Control



B. Theory Translation

- Consider the following reference model for the **speed controller** in a European ETCS high-speed train
- There the inputs are
 - allowed speed
 - actual speed
- These are floating point values that **cannot be enumerated** as it was possible when testing against an FSM reference model
- Can we develop a **complete test suite** despite the fact that the input domain is conceptually infinite?

Onboard Main Controller – Reference Model as SysML State Machine



Allowed maximal speed v_m in $[0, 400]$
Current speed v in $[0, 400]$

B. Theory Translation

- Consider different **semantic domains** with their conformance relations
 - Finite state machines – I/O-equivalence
 - SysML state machines – I/O-equivalence
 - Fix a **signature** in each domain
 - Sig_1 – SysML state machines over fixed I/O variables with real-valued inputs and discrete outputs
 - Sig_2 – FSMs over fixed discrete I/O-alphabet

B. Theory Translation

- Create a **model map** T from sub-domain of Sig_1 to Sig_2

$$T : Dom_1 \rightarrow Sig_2;$$
$$Dom_1 \subseteq Sig_1$$

- Create a **test case map** T^* from test cases of Sig_2 to test cases of Sig_1

$$T^* : TC(Sig_2) \rightarrow TC(Sig_1)$$

- Prove the **satisfaction condition**

$$(T, T^*)$$

Satisfaction Condition

Condition 1. The model map is compatible with the conformance relations

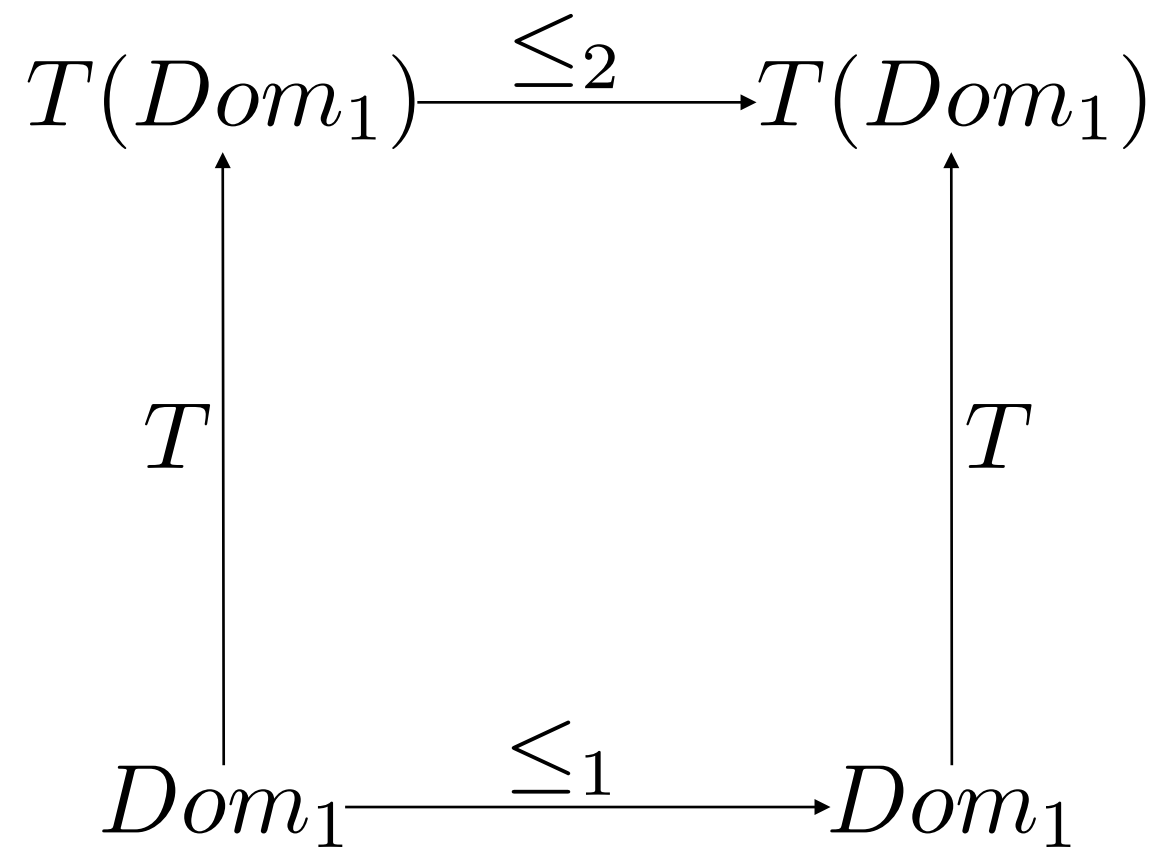
$$\forall \mathcal{S}, \mathcal{S}' \in Dom_1 : \mathcal{S}' \leq_1 \mathcal{S} \Leftrightarrow T(\mathcal{S}') \leq_2 T(\mathcal{S})$$

Condition 2. Model map and test case map preserve the pass relationship

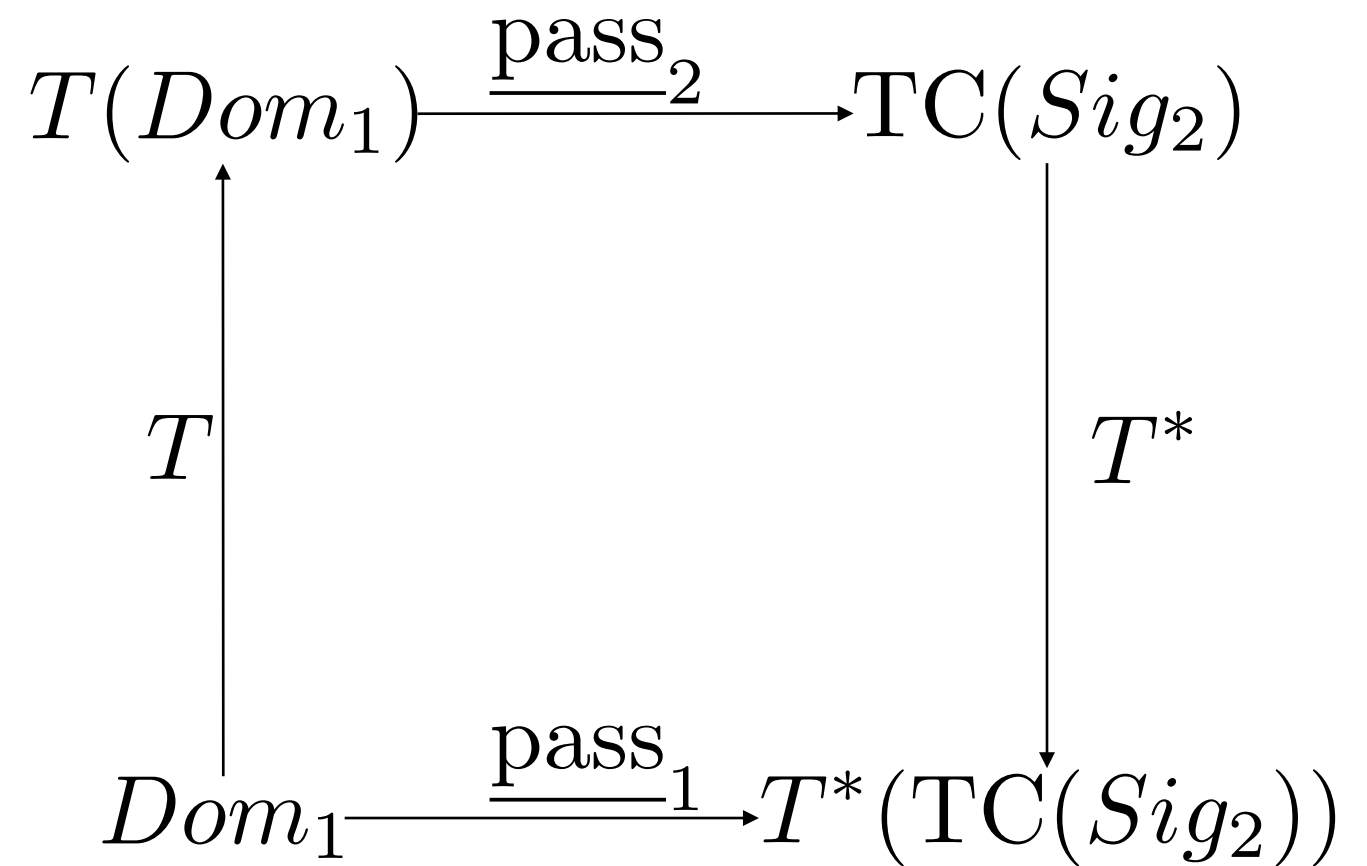
$$\forall \mathcal{S} \in Dom_1, U \in TC(Sig_2) : T(\mathcal{S}) \underline{pass}_2 U \Leftrightarrow \mathcal{S} \underline{pass}_1 T^*(U)$$

Satisfaction condition,
reflected by commuting diagrams

Condition 1

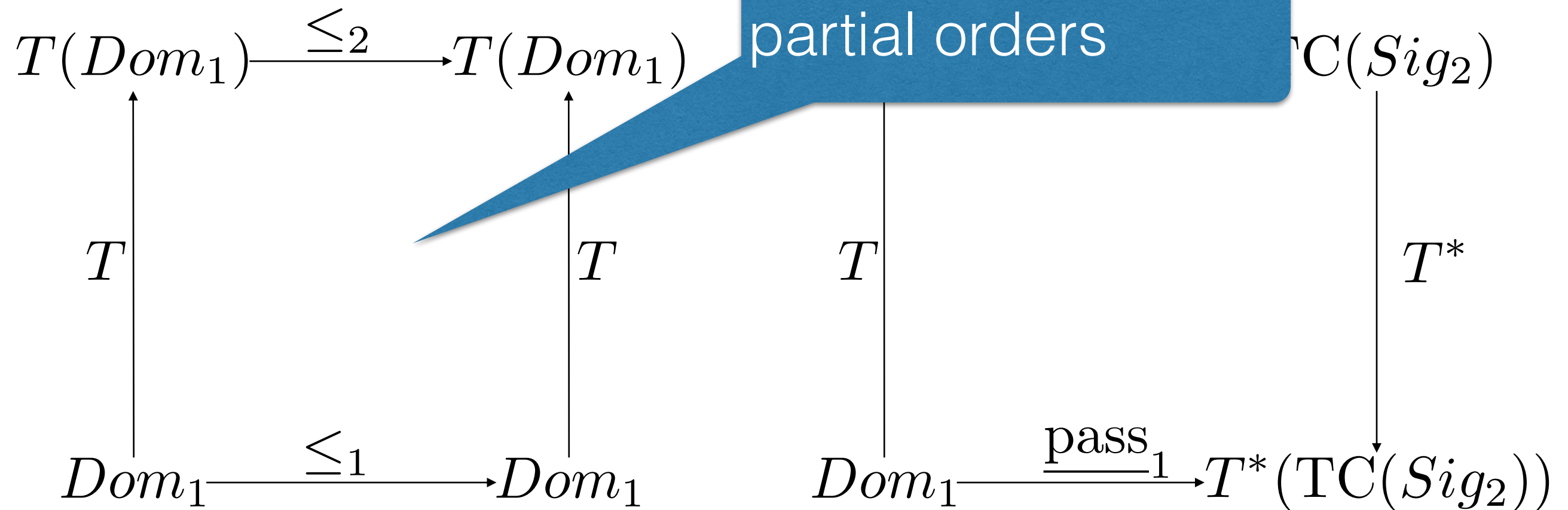


Condition 2



Satisfaction condition,
reflected by commuting diagrams

Condition 1



General Theorem for Translation of Testing Theories

Theorem 1. Suppose (T, T^*) exist and fulfil the satisfaction condition. Then every complete testing theory established in Sig_2 induces a likewise complete testing theory on Sig_1 .

Theory Translation

- **Theorem 2.** Every complete FSM testing theory for
 - I/O-equivalence or
 - reduction (I/O-sequence containment)

induces a complete **equivalence class testing theory** with analogous conformance relations for SysML state machines with **infinite input domains**, bounded nondeterminism, and finite internal state and finite outputs

Wen-ling Huang, Jan Peleska:
Complete Model-Based Equivalence Class Testing.
J Softw Tools Technol Transfer 18, No. 3, pp. 265-283, 2016
DOI 10.1007/s10009-014-0356-8., 2016

Wen-ling Huang, Jan Peleska:
Complete Model-Based Equivalence Class Testing for Nondeterministic Systems.
Under review in Formal Aspects of Computing, 2016



Theory

More details are explained in Wen-ling Huang's lecture 'Testing infinite state systems - mathematical foundations and concrete algorithms' on Thursday, 2016-05-26, ST527, 15:20 – 17:00

- **Theorem 2.** Every complete FSM testing theory for
 - I/O-equivalence or
 - reduction (I/O-sequence containment)

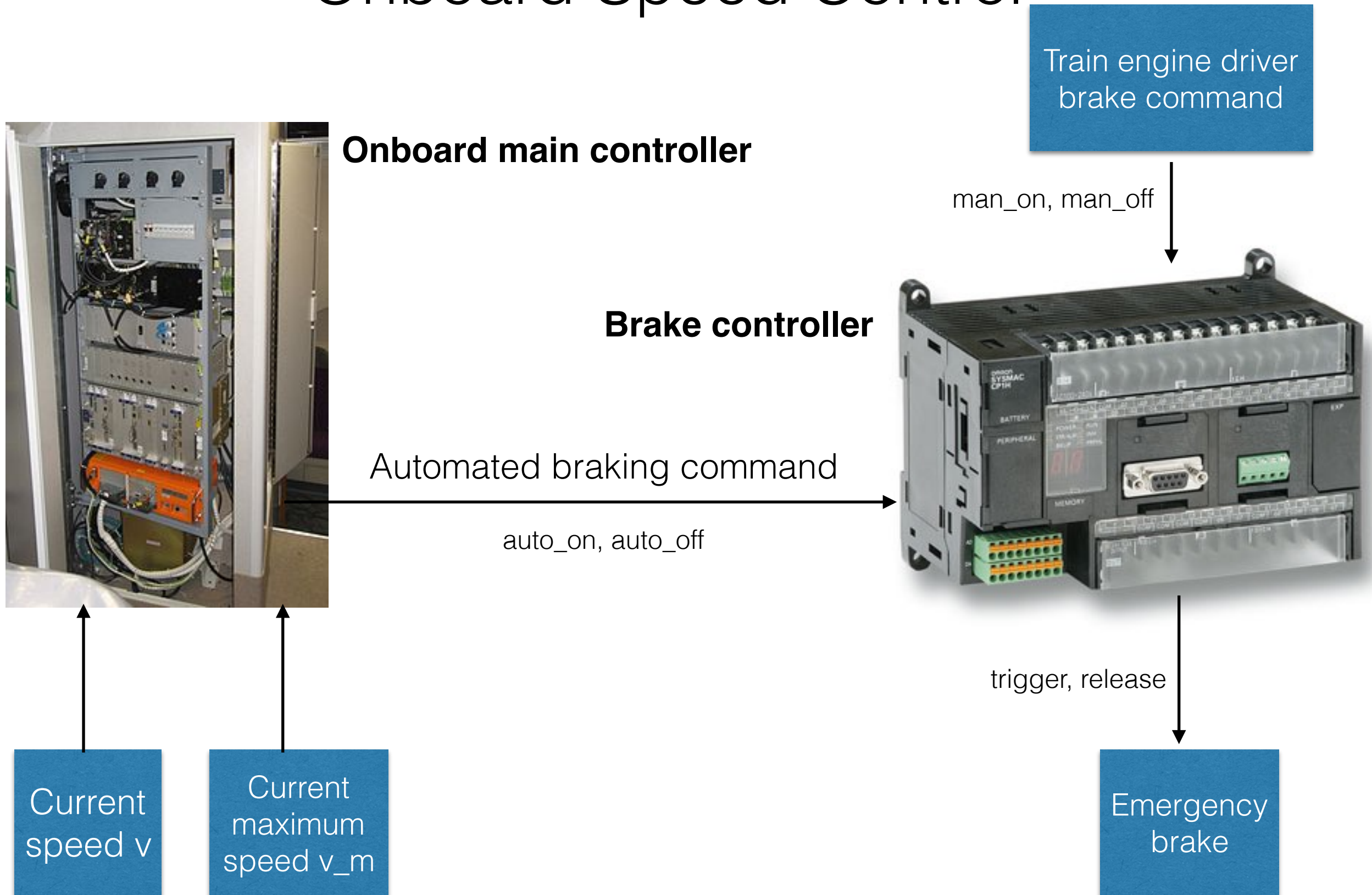
induces a complete **equivalence class partition testing theory** with analogous conformance relations for SysML state machines with **infinite input domains**, bounded nondeterminism, and finite internal state and finite outputs

Wen-ling Huang, Jan Peleska:
Complete Model-Based Equivalence Class Testing.
J Softw Tools Technol Transfer 18, No. 3, pp. 265-283, 2016
DOI 10.1007/s10009-014-0356-8., 2016

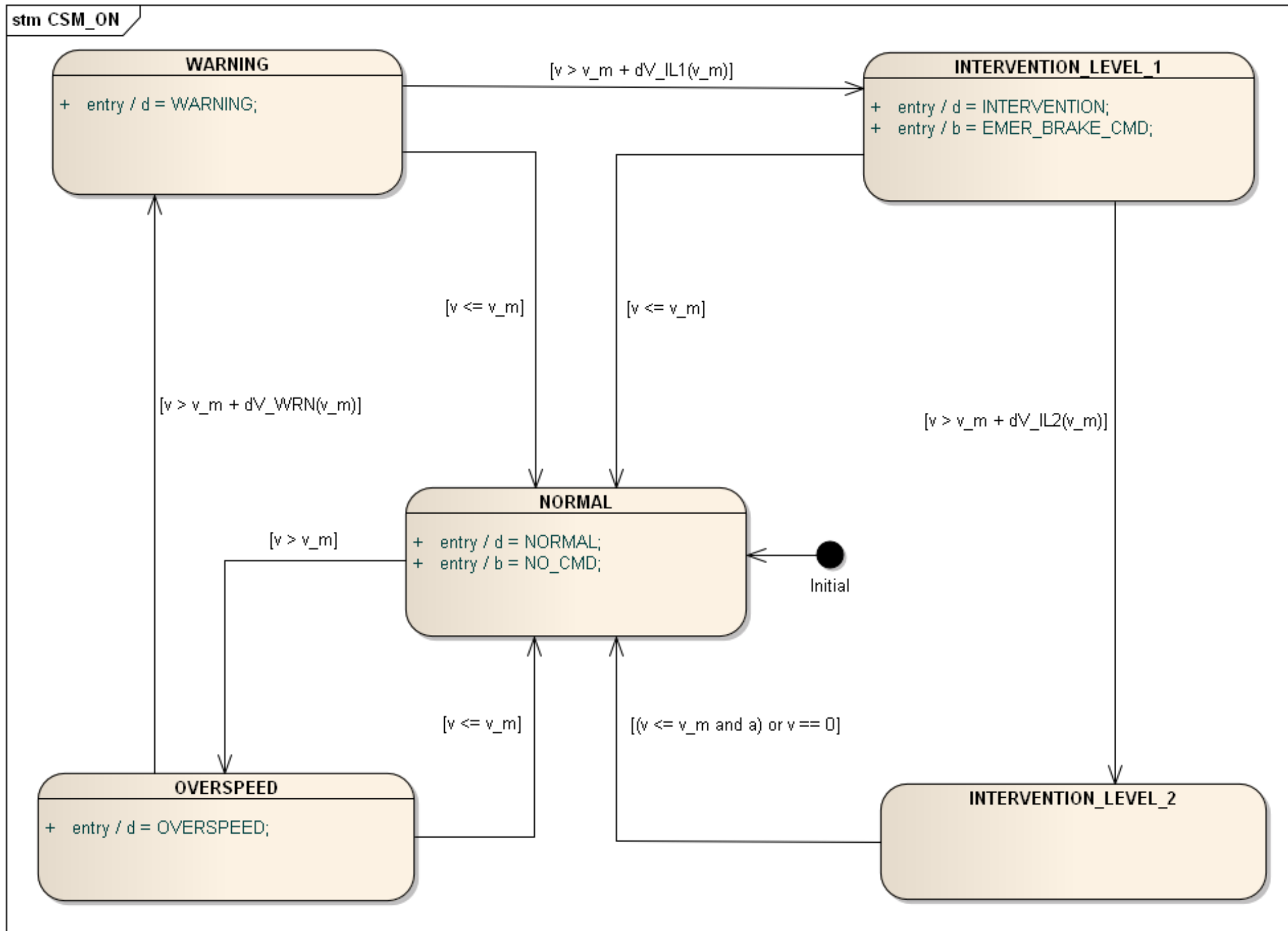
Wen-ling Huang, Jan Peleska:
Complete Model-Based Equivalence Class Testing for Nondeterministic Systems.
Under review in Formal Aspects of Computing, 2016



Recall: Application Scenario – Train Onboard Speed Control



Onboard Main Controller – Reference Model as SysML State Machine



Allowed maximal speed v_m in $[0, 400]$
Current speed v in $[0, 400]$

Application of Theorem 2

1. Calculate **input equivalence classes**
2. **Map speed monitor model to FSM** with
 1. input equivalence classes as input alphabet
 2. original discrete outputs as output alphabet
3. Use W-Method or similar method to **create complete FSM test suite**
4. **Translate FSM test suite** to concrete test suite for speed monitor

Calculate equivalence classes – Step 1.

Determine transition

relation of the SysML state machine

$$\mathcal{R} \equiv \bigvee_{i \in \text{IDX}} (\alpha_i \wedge (m, y) = (m_i, y_i) \wedge (m', y') = (m_i, y_i))$$
$$\vee \bigvee_{(i,j) \in J} (g_{i,j} \wedge (m, y) = (m_i, y_i) \wedge (m', y') = (m_j, y_j))$$

$$A_i = \{s \in S \mid s(\alpha_i) \wedge s(m) = m_i \wedge s(y) = y_i\}$$

$$g_{1,2} \equiv v_m < v \wedge v \leq v_m + dV_{\text{WRN}}(v_m)$$

$$g_{1,3} \equiv v_m + dV_{\text{WRN}}(v_m) < v \leq v_m + dV_{\text{IL1}}(v_m)$$

$$g_{1,4} \equiv v_m + dV_{\text{IL1}}(v_m) < v \leq v_m + dV_{\text{IL2}}(v_m)$$

$$g_{1,5} \equiv v_m + dV_{\text{IL2}}(v_m) < v$$

$$g_{2,3} \equiv g_{1,3}$$

$$g_{2,4} \equiv g_{3,4} \equiv g_{1,4}$$

$$g_{2,5} \equiv g_{3,5} \equiv g_{4,5} \equiv g_{1,5}$$

$$g_{2,1} \equiv v \leq v_m$$

$$g_{3,1} \equiv g_{4,1} \equiv g_{2,1}$$

$$g_{5,1} \equiv v = 0 \vee (v \leq v_m \wedge a = 1)$$

Calculate equivalence classes – Step 2.

Identify logical formulas for each input equivalence class

$$\Phi_1 \equiv g_{1,1} \wedge g_{2,1} \wedge g_{3,1} \wedge g_{4,1} \wedge g_{5,5}$$

$$\equiv 0 < v \leq v_m \wedge a = 0$$

$$\Phi_2 \equiv g_{1,1} \wedge g_{2,1} \wedge g_{3,1} \wedge g_{4,1} \wedge g_{5,1}$$

$$\equiv v = 0 \vee (v \leq v_m \wedge a = 1)$$

$$\Phi_3 \equiv g_{1,2} \wedge g_{2,2} \wedge g_{3,3} \wedge g_{4,4} \wedge g_{5,5}$$

$$\equiv v_m < v \leq v_m + dV_{\text{WRN}}(v_m)$$

$$\Phi_4 \equiv g_{1,3} \wedge g_{2,3} \wedge g_{3,3} \wedge g_{4,4} \wedge g_{5,5}$$

$$\equiv v_m + dV_{\text{WRN}}(v_m) < v \leq v_m + dV_{\text{IL1}}(v_m)$$

$$\Phi_5 \equiv g_{1,4} \wedge g_{2,4} \wedge g_{3,4} \wedge g_{4,4} \wedge g_{5,5}$$

$$\equiv v_m + dV_{\text{IL1}}(v_m) < v \leq v_m + dV_{\text{IL2}}(v_m)$$

$$\Phi_6 \equiv g_{1,5} \wedge g_{2,5} \wedge g_{3,5} \wedge g_{4,5} \wedge g_{5,5}$$

$$\equiv v_m + dV_{\text{IL2}}(v_m) < v$$

Define

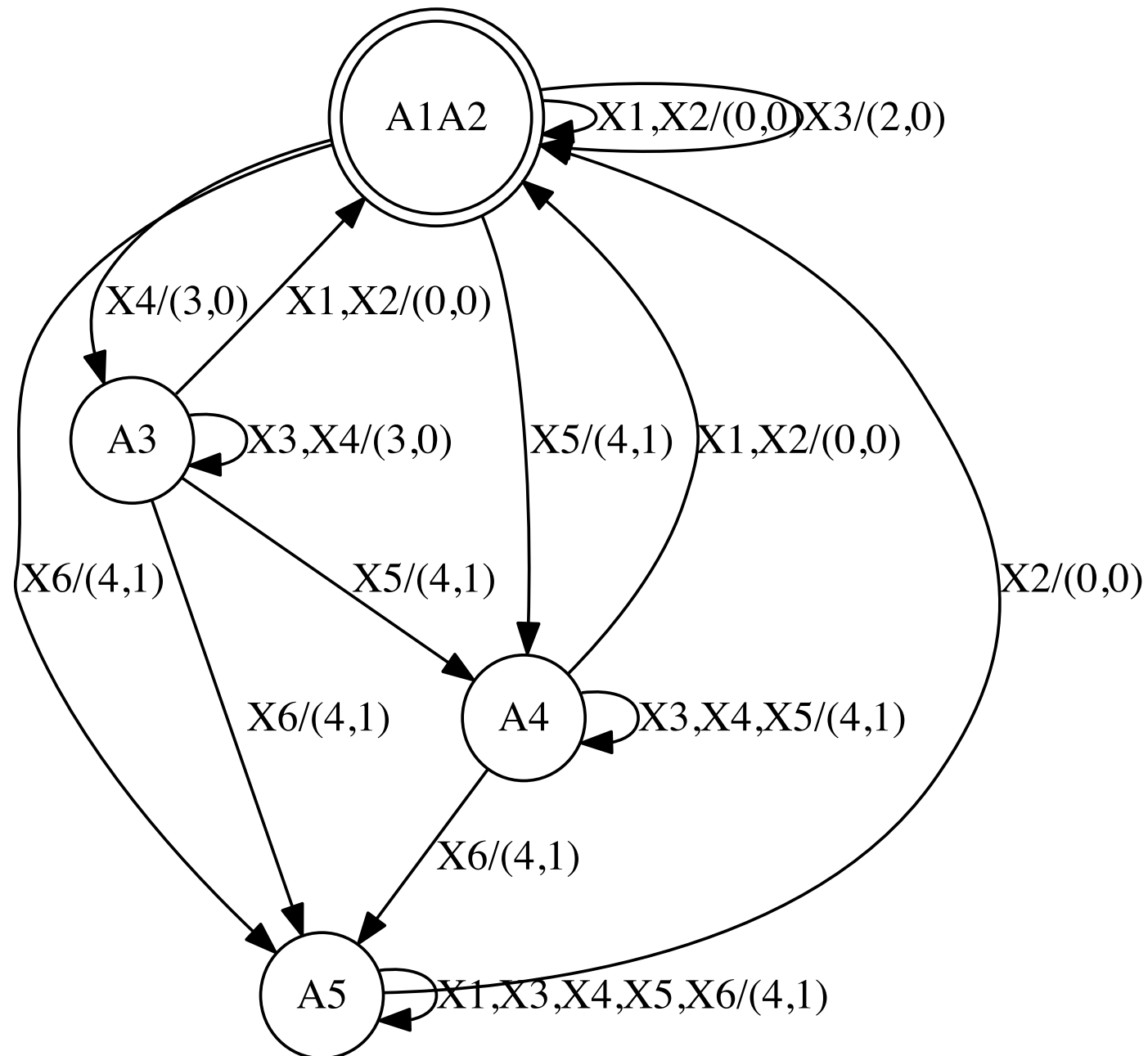
$$X_i = \{(v, v_m, a) \in I \mid (v, v_m, a) \models \Phi_i\} \quad i = 1, \dots, 6$$

$\mathcal{I} = \{X_1, X_2, \dots, X_6\}$. Then \mathcal{I} is an IECP for the CSM

Map SysML model of the speed monitor to FSM

FSM inputs: input equivalence classes X_i

FSM outputs: finite outputs as defined for SysML model



Symbolic test cases resulting from W-Method

1. $x_4 \cdot x_1 \cdot x_3 / (3, 0) \cdot (0, 0) \cdot (2, 0)$
2. $x_4 \cdot x_1 \cdot x_1 / (3, 0) \cdot (0, 0) \cdot (0, 0)$
3. $x_4 \cdot x_2 \cdot x_3 / (3, 0) \cdot (0, 0) \cdot (2, 0)$
4. $x_4 \cdot x_2 \cdot x_1 / (3, 0) \cdot (0, 0) \cdot (0, 0)$
5. $x_4 \cdot x_3 \cdot x_3 / (3, 0) \cdot (3, 0) \cdot (3, 0)$
6. $x_4 \cdot x_3 \cdot x_1 / (3, 0) \cdot (3, 0) \cdot (0, 0)$
7. $x_4 \cdot x_4 \cdot x_3 / (3, 0) \cdot (3, 0) \cdot (3, 0)$
8. $x_4 \cdot x_4 \cdot x_1 / (3, 0) \cdot (3, 0) \cdot (0, 0)$
9. $x_4 \cdot x_5 \cdot x_3 / (3, 0) \cdot (4, 1) \cdot (4, 1)$
10. $x_4 \cdot x_5 \cdot x_1 / (3, 0) \cdot (4, 1) \cdot (0, 0)$
11. $x_4 \cdot x_6 \cdot x_3 / (3, 0) \cdot (4, 1) \cdot (4, 1)$
12. $x_4 \cdot x_6 \cdot x_1 / (3, 0) \cdot (4, 1) \cdot (4, 1)$
13. $x_5 \cdot x_1 \cdot x_3 / (4, 1) \cdot (0, 0) \cdot (2, 0)$
14. $x_5 \cdot x_1 \cdot x_1 / (4, 1) \cdot (0, 0) \cdot (0, 0)$
15. $x_5 \cdot x_2 \cdot x_3 / (4, 1) \cdot (0, 0) \cdot (2, 0)$
16. $x_5 \cdot x_2 \cdot x_1 / (4, 1) \cdot (0, 0) \cdot (0, 0)$
17. $x_5 \cdot x_3 \cdot x_3 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
18. $x_5 \cdot x_3 \cdot x_1 / (4, 1) \cdot (4, 1) \cdot (0, 0)$
19. $x_5 \cdot x_4 \cdot x_3 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
20. $x_5 \cdot x_4 \cdot x_1 / (4, 1) \cdot (4, 1) \cdot (0, 0)$
21. $x_5 \cdot x_5 \cdot x_3 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
22. $x_5 \cdot x_5 \cdot x_1 / (4, 1) \cdot (4, 1) \cdot (0, 0)$
23. $x_5 \cdot x_6 \cdot x_3 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
24. $x_5 \cdot x_6 \cdot x_1 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
25. $x_6 \cdot x_1 \cdot x_3 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
26. $x_6 \cdot x_1 \cdot x_1 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
27. $x_6 \cdot x_2 \cdot x_3 / (4, 1) \cdot (0, 0) \cdot (2, 0)$
28. $x_6 \cdot x_2 \cdot x_1 / (4, 1) \cdot (0, 0) \cdot (0, 0)$
29. $x_6 \cdot x_3 \cdot x_3 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
30. $x_6 \cdot x_3 \cdot x_1 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
31. $x_6 \cdot x_4 \cdot x_3 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
32. $x_6 \cdot x_4 \cdot x_1 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
33. $x_6 \cdot x_5 \cdot x_3 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
34. $x_6 \cdot x_5 \cdot x_1 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
35. $x_6 \cdot x_6 \cdot x_3 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
36. $x_6 \cdot x_6 \cdot x_1 / (4, 1) \cdot (4, 1) \cdot (4, 1)$
37. $x_1 \cdot x_3 / (0, 0) \cdot (2, 0)$
38. $x_1 \cdot x_1 / (0, 0) \cdot (0, 0)$
39. $x_2 \cdot x_3 / (0, 0) \cdot (2, 0)$
40. $x_2 \cdot x_1 / (0, 0) \cdot (0, 0)$
41. $x_3 \cdot x_3 / (2, 0) \cdot (2, 0)$
42. $x_3 \cdot x_1 / (2, 0) \cdot (0, 0)$

Symbolic test cases resulting from W-Method

1.	$X4.X1.X3 / (2, 0) . (0, 0) . (2, 0)$	21.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
2.	$X4.X1.X3 / (4, 1) . (4, 1) . (0, 0)$	22.	$X5.X5.X2 / (4, 1) . (4, 1) . (0, 0)$
3.	$X4.X2.X3 / (4, 1) . (4, 1) . (4, 1)$	23.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
4.	$X4.X2.X3 / (4, 1) . (4, 1) . (4, 1)$	24.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
5.	$X4.X3.X3 / (4, 1) . (4, 1) . (4, 1)$	25.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
6.	$X4.X3.X3 / (4, 1) . (4, 1) . (4, 1)$	26.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
7.	$X4.X4.X3 / (4, 1) . (0, 0) . (2, 0)$	27.	$X5.X5.X2 / (4, 1) . (0, 0) . (2, 0)$
8.	$X4.X4.X3 / (4, 1) . (0, 0) . (0, 0)$	28.	$X5.X5.X2 / (4, 1) . (0, 0) . (0, 0)$
9.	$X4.X5.X3 / (4, 1) . (4, 1) . (4, 1)$	29.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
10.	$X4.X5.X3 / (4, 1) . (4, 1) . (4, 1)$	30.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
11.	$X4.X6.X3 / (4, 1) . (4, 1) . (4, 1)$	31.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
12.	$X4.X6.X3 / (4, 1) . (4, 1) . (4, 1)$	32.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
13.	$X5.X1.X3 / (4, 1) . (4, 1) . (4, 1)$	33.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
14.	$X5.X1.X3 / (4, 1) . (4, 1) . (4, 1)$	34.	$X5.X5.X2 / (4, 1) . (4, 1) . (4, 1)$
15.	$X5.X2.X3 / (4, 1) . (0, 0) . (2, 0)$	35.	$X6.X6.X3 / (4, 1) . (4, 1) . (4, 1)$
16.	$X5.X2.X1 / (4, 1) . (0, 0) . (0, 0)$	36.	$X6.X6.X1 / (4, 1) . (4, 1) . (4, 1)$
17.	$X5.X3.X3 / (4, 1) . (4, 1) . (4, 1)$	37.	$X1.X3 / (0, 0) . (2, 0)$
18.	$X5.X3.X1 / (4, 1) . (4, 1) . (0, 0)$	38.	$X1.X1 / (0, 0) . (0, 0)$
19.	$X5.X4.X3 / (4, 1) . (4, 1) . (4, 1)$	39.	$X2.X3 / (0, 0) . (2, 0)$
20.	$X5.X4.X1 / (4, 1) . (4, 1) . (0, 0)$	40.	$X2.X1 / (0, 0) . (0, 0)$
		41.	$X3.X3 / (2, 0) . (2, 0)$
		42.	$X3.X1 / (2, 0) . (0, 0)$

Symbolic means that concrete test data still has to be selected from each X_i when it is referenced in a test case

This can be done automatically using a mathematical constraint solver (**SMT-solver**)

Combination With Random and Boundary Value Testing

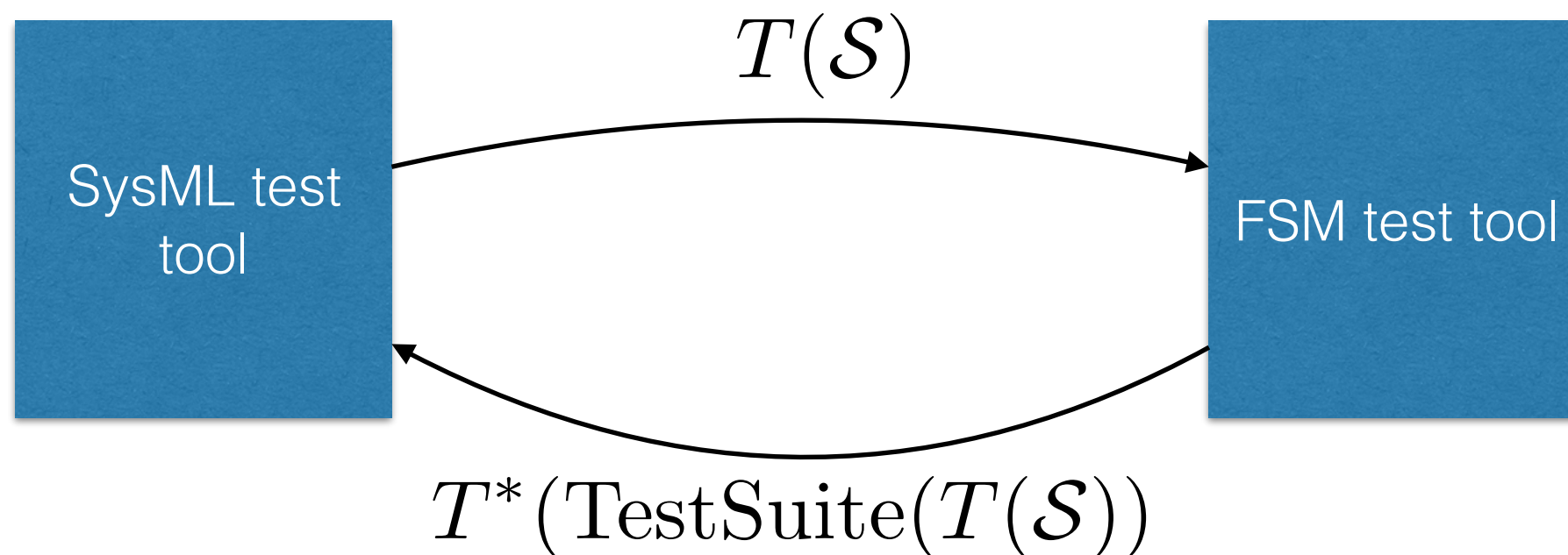
- Instead of always using the same representative of each input class representative, **select a random value of this class**, whenever it is used in the test case – combine this technique with **boundary value tests**
- Completeness is still guaranteed for SUTs inside the fault domain
- **For SUTs outside the fault domain, the test strength is significantly increased**

Summary of the Benefits

- A new complete testing strategy for systems with infinite input domains and finite internal states and finite outputs
- Effectively implementable in model-based testing tools – fully automated
- Significantly **higher test strength** compared to heuristic test strategies
- Significant **reduction of test effort** in application domains where the testing is very costly: railway interlocking systems

Summary of the Benefits

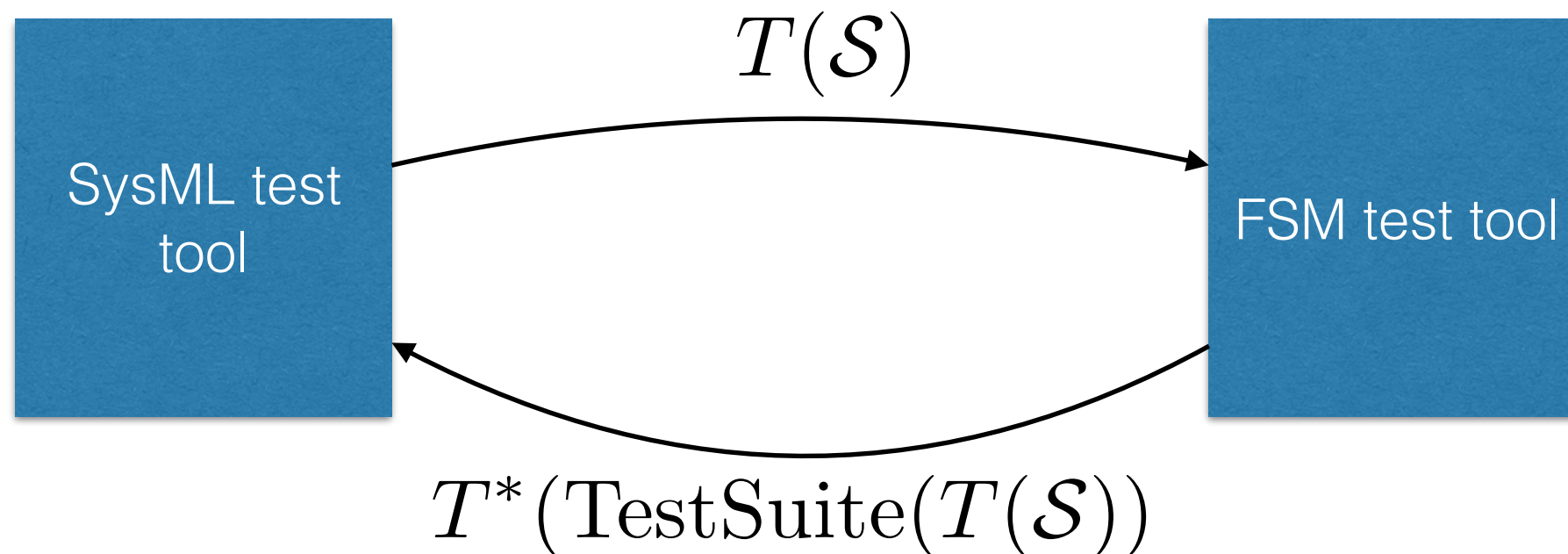
- When building a new tool for model-based testing of SysML state machines (with infinite input domains), the test case generation can be performed by an existing tool implementing these algorithms for FSMs



Summary of the Benefits

- When building a new tool for model-based testing of SysML state machines (with conceptually infinite input domains), the test performed by an existing algorithms for FSMs

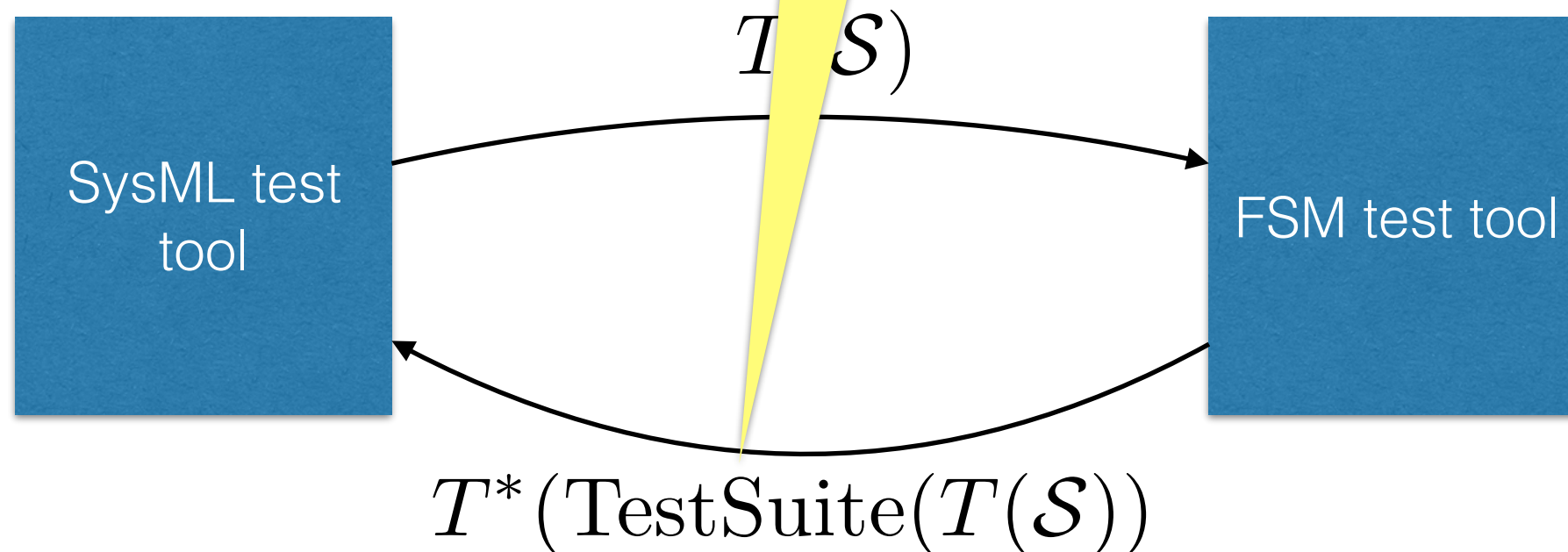
SysML test tool performs FSM abstraction and sends it to FSM test tool



Summary of the Benefits

- When building a new tool for model-based testing of SysML state machines (with conceptually infinite input domains), the test performed by an existing algorithms for FSMs

FSM tool generates complete test suite and sends the translated result to SysML test tool



Theory Meets Innovation –

C. Theory translation for property checking: a complete testing strategy for checking safety properties

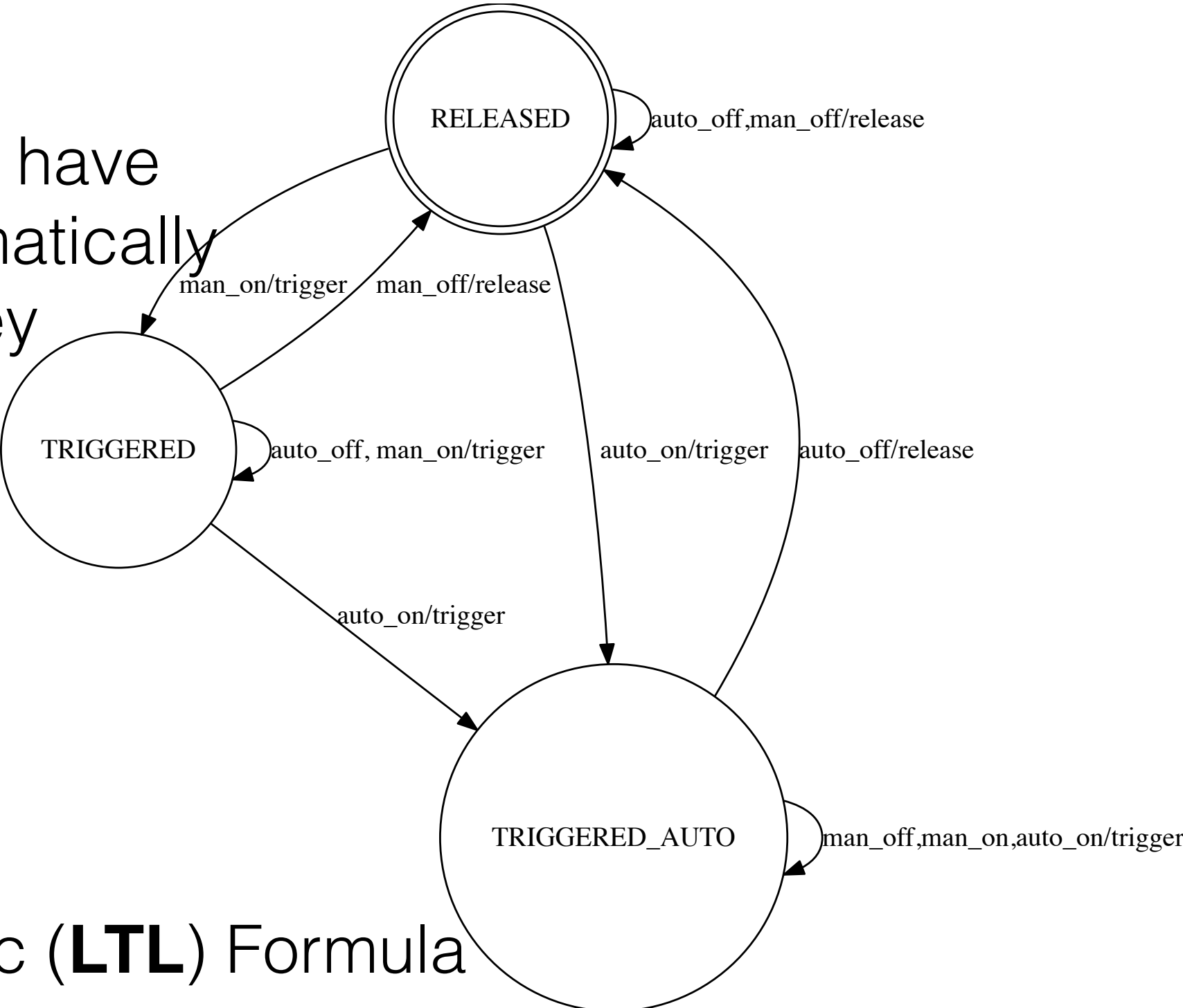
C. Theory Translation

- **Objective.** Property checking by means of complete testing strategies
- **Property checking.** Instead of checking the implementation with respect to conformance (e.g. I/O-equivalence) with a reference model, we just want to ensure that the implementation fulfils a specific property, for example, a **safety property**

Example. Brake controller

Safety property.

Whenever the brakes have been triggered automatically by event *auto_on*, they can only be released by the *auto_off* command



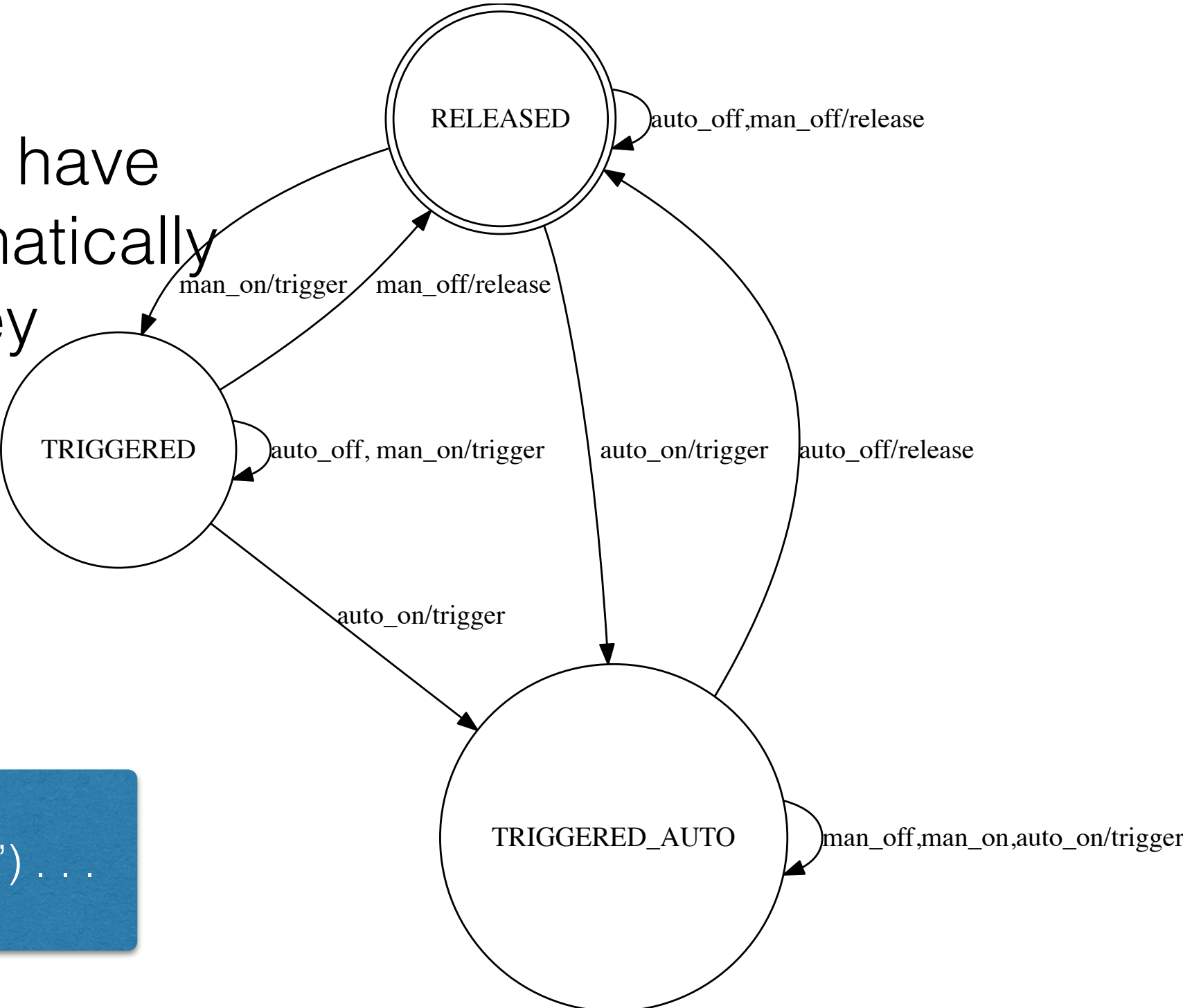
Linear Temporal Logic (**LTL**) Formula

$$G(\text{auto_on} \Rightarrow X(\text{trigger } W \text{ auto_off}))$$

Example. Brake controller

Safety property.

Whenever the brakes have been triggered automatically by event *auto_on*, they can only be released by the *auto_off* command



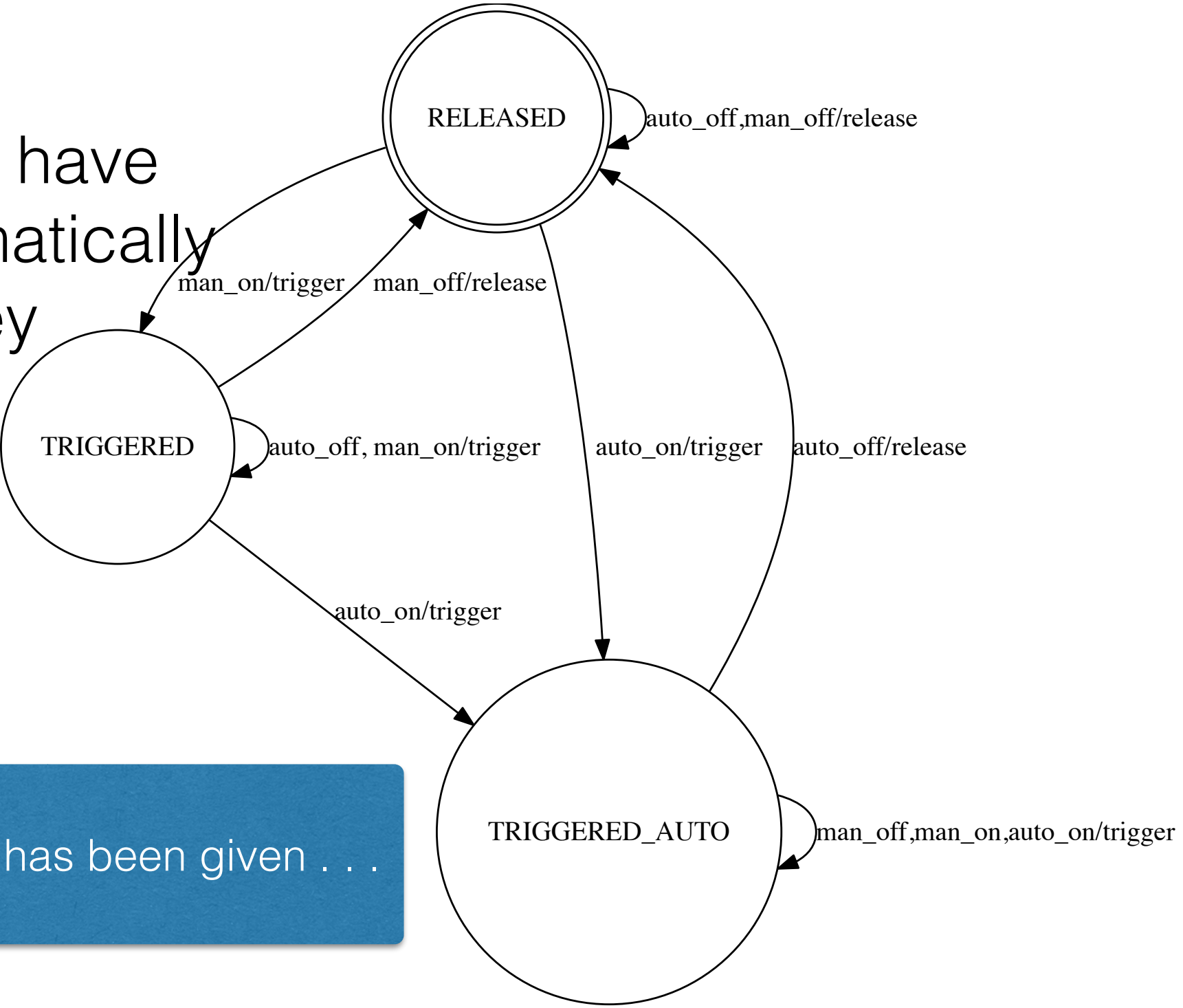
Whenever (“Globally”) . . .

$$G(\text{auto_on} \Rightarrow X(\text{trigger } W \text{ auto_off}))$$

Example. Brake controller

Safety property.

Whenever the brakes have been triggered automatically by event *auto_on*, they can only be released by the *auto_off* command



... command *auto_on* has been given ...

$$G(\text{auto_on} \Rightarrow X(\text{trigger } W \text{ auto_off}))$$

Example. Brake controller

Safety property.

Whenever the brakes have been triggered automatically by event *auto_on*, they can only be released by the *auto_off* command



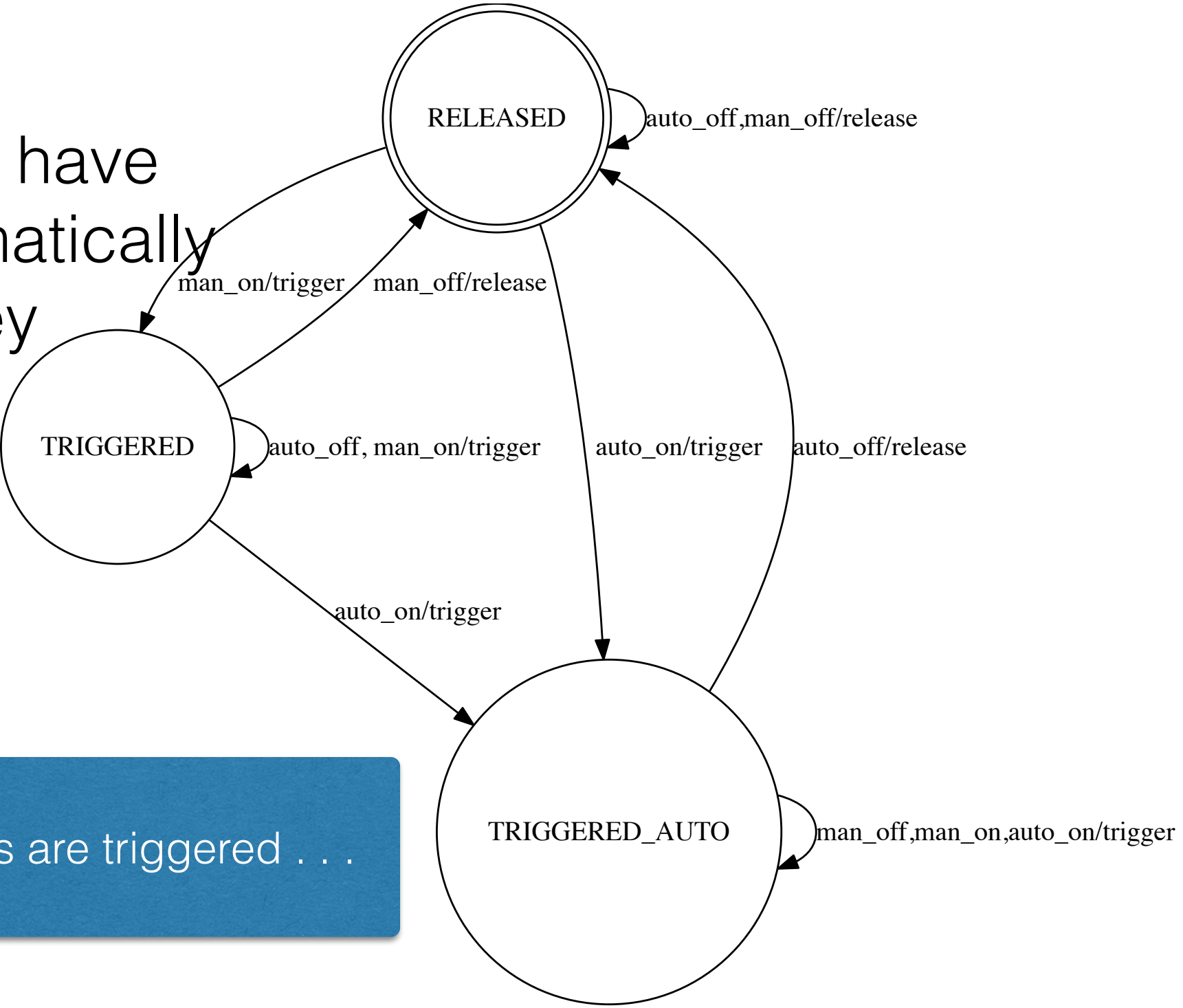
... then in the next step ...

$$G(\text{auto_on} \Rightarrow X(\text{trigger } W \text{ auto_off}))$$

Example. Brake controller

Safety property.

Whenever the brakes have been triggered automatically by event *auto_on*, they can only be released by the *auto_off* command



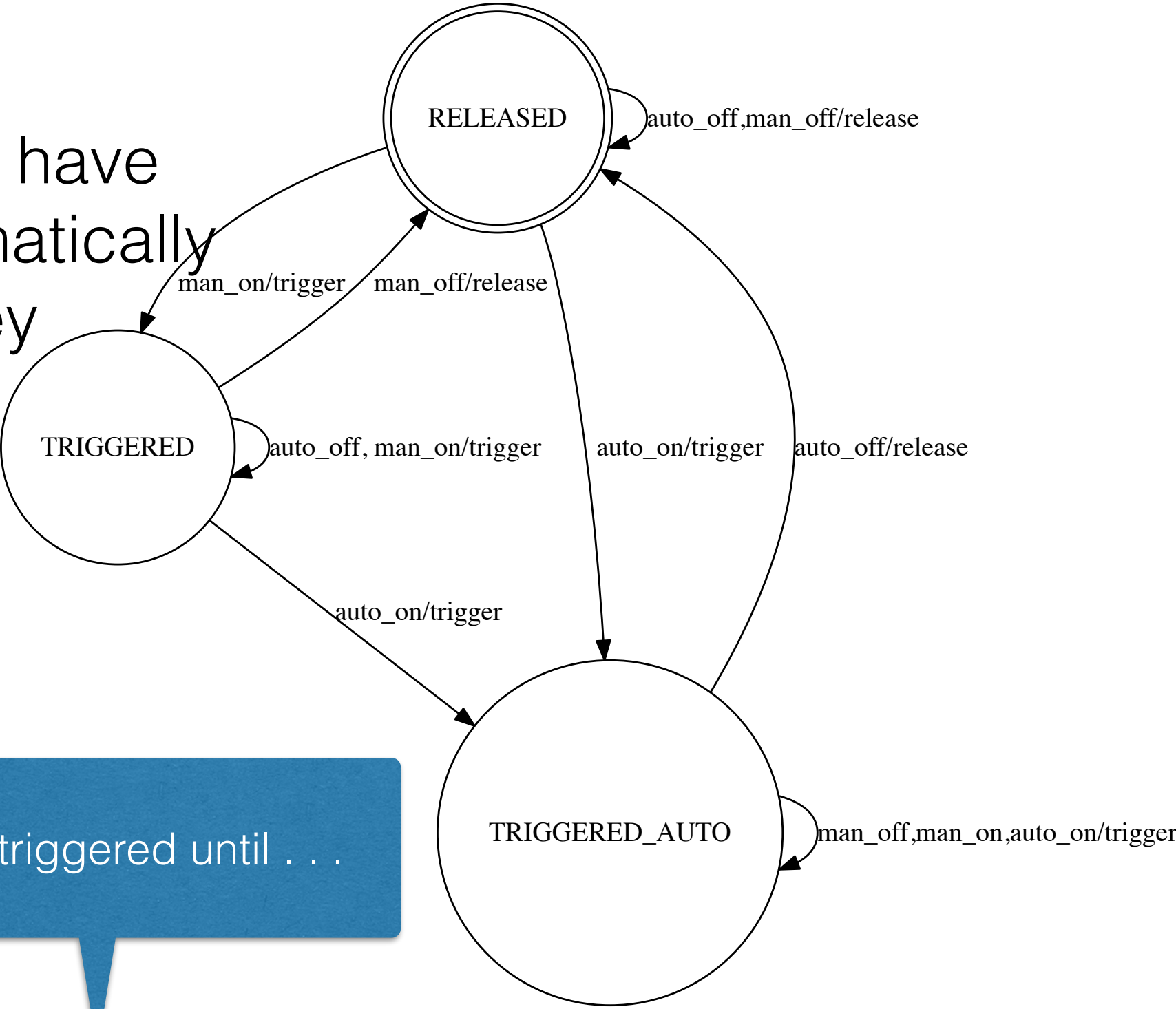
... the brakes are triggered ...

$$G(\text{auto_on} \Rightarrow X(\text{trigger } W \text{ auto_off}))$$

Example. Brake controller

Safety property.

Whenever the brakes have been triggered automatically by event *auto_on*, they can only be released by the *auto_off* command



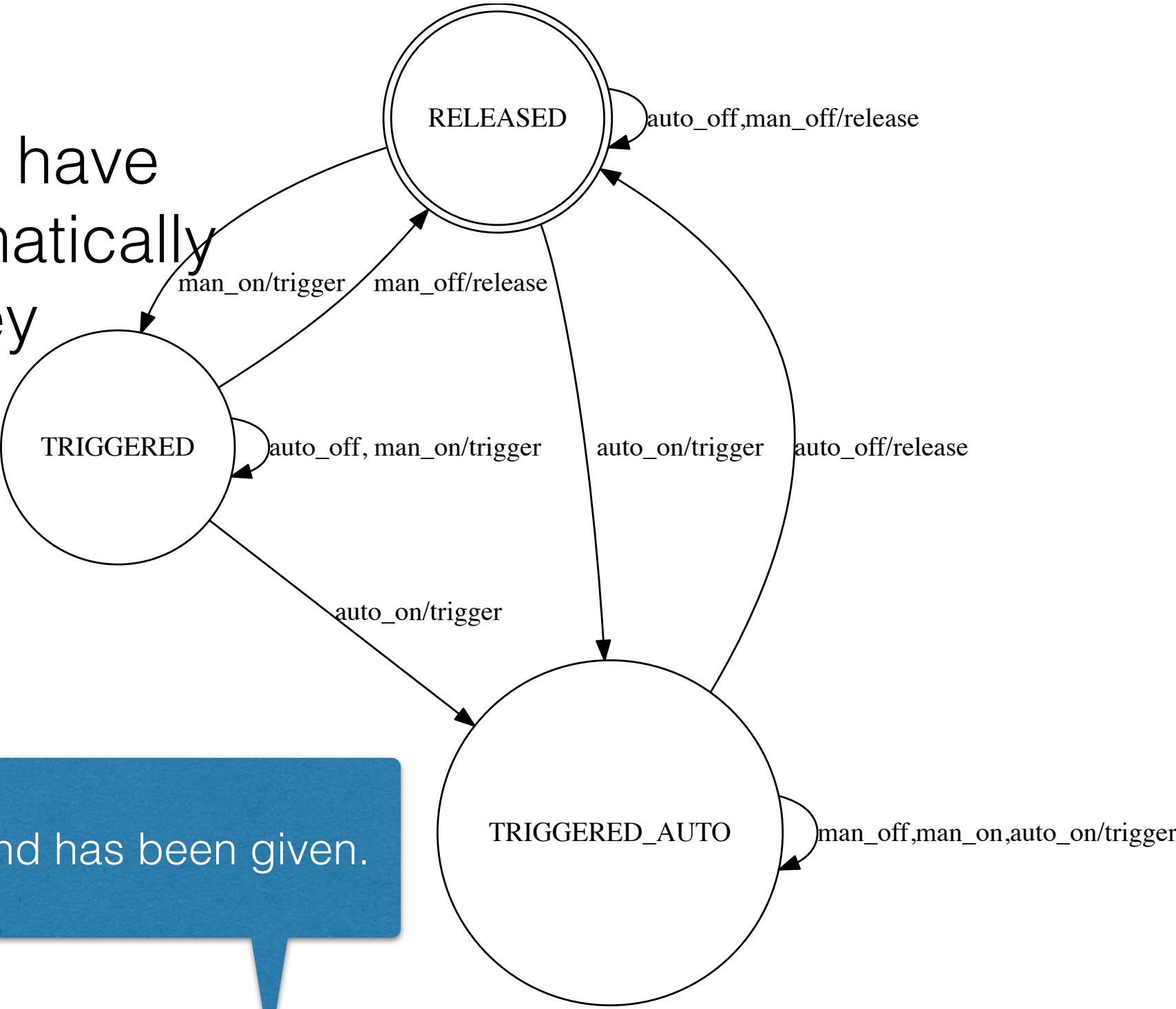
... and stay triggered until ...

$$G(\text{auto_on} \Rightarrow X(\text{trigger } W \text{ auto_off}))$$

Example. Brake controller

Safety property.

Whenever the brakes have been triggered automatically by event *auto_on*, they can only be released by the *auto_off* command



... the auto_off command has been given.

$$G(\text{auto_on} \Rightarrow X(\text{trigger } W \text{ auto_off}))$$

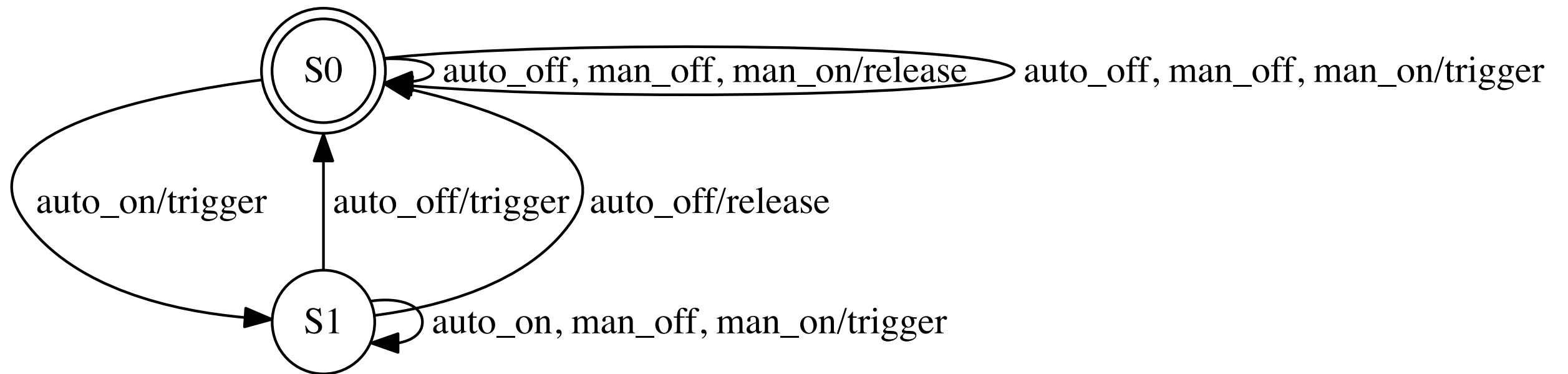
Do we need a new method?

Not required, because we can proceed as follows

- Generate the “most liberal” FSM that “just fulfils” the required safety property, but behaves “chaotically” otherwise – this is used as **reference model**
- Apply a complete test method that allows to prove that the implementation is a reduction of the reference model
 - **Reduction**. The implementation only performs I/O-sequences that can also be performed by the reference model
 - Reduction implies that **every safety property fulfilled by the reference model is also fulfilled by the implementation**

The **reference model** “just fulfilling”

$G(\text{auto_on} \Rightarrow X(\text{trigger } W \text{ auto_off}))$



C. Theory Translation: Can we extend this concept to property checking of more complex systems than those modelled by FSMs?

Yes, of course! We use again the theory translation method introduced above – see the recipe next page

- This recipe is applicable (as before) to systems with
 - input variables over infinite domains
 - finite internal state variables
 - output variables with finite domains
- . . . and to LTL safety formulas

1. Take the reference model \mathcal{S} of the system and calculate its input equivalence classes $\longrightarrow \mathcal{I} = \{X_1, \dots, X_k\}$
2. Specify the LTL safety formula $\longrightarrow \varphi$
3. Extract all atomic propositions from φ : $\longrightarrow AP = \{p_1, \dots, p_n\}$
4. Decompose all $p_i \in AP$ in such a way that the new atomic propositions only contain free variables in either I , M , or O : $\longrightarrow \underline{AP} = \{q_1, \dots, q_\ell\} = \underline{AP}_I \cup \underline{AP}_M \cup \underline{AP}_O$
5. Refine \mathcal{I} by \underline{AP} : $\longrightarrow \underline{\mathcal{I}}$
6. Translate φ to the most nondeterministic FSM M_φ satisfying φ : \longrightarrow input alphabet \underline{AP}_I , output alphabet \underline{AP}_O
7. Generate FSM test suite from M_φ that is complete for showing that an implementation in the fault domain is a reduction of M_φ
8. Translate FSM test suite to test suite for implementation \mathcal{S}'
9. If \mathcal{S}' passes all tests, it fulfils the property $\longrightarrow \mathcal{S}' \models \varphi$

1. Take the reference model \mathcal{S} of the system and calculate its input equivalence classes $\longrightarrow \mathcal{I} = \{X_1, \dots, X_k\}$
2. Specify the LTL safety formula φ — These details are explained in Wen-ling Huang's 3rd lecture 'Property checking of safety-critical systems - mathematical foundations and concrete algorithms'
3. Extract all atomic propositions from φ
4. Decompose all $p_i \in AP$ in such a way that they only contain free variables in either $\underline{AP}_I \cup \underline{AP}_M \cup \underline{AP}_O$ Friday, 2016-05-27, 10:20 – 12:00, ST527
5. Refine \mathcal{I} by \underline{AP} : $\longrightarrow \underline{\mathcal{I}}$
6. Translate φ to the most nondeterministic FSM M_φ satisfying φ : \longrightarrow input alphabet \underline{AP}_I , output alphabet \underline{AP}_O
7. Generate FSM test suite from M_φ that is complete for showing that an implementation in the fault domain is a reduction of M_φ
8. Translate FSM test suite to test suite for implementation \mathcal{S}'
9. If \mathcal{S}' passes all tests, it fulfils the property $\longrightarrow \mathcal{S}' \models \varphi$

Conclusion

Summarising, we have illustrated how . . .

- . . . the theory of complete testing strategies helps in **uncovering more errors** and is effective in situations where **certification considerations** require to justify the suitability of test suites
- . . . the translation of testing theories helps to design new testing strategies, in particular for **equivalence class testing** and **property testing**
- . . . **automation is necessary** to exploit the benefits of these new methods: you could never design these tests by hand!

Verified Systems International was awarded the runner-up trophy of the European Innovation Radar Innovation Prize 2015 for integrating the equivalence class testing theory developed by Wen-ling Huang and Jan Peleska into their test automation product RT-Tester



Further Reading

1. Publications of Jan Peleska, Wen-ling Huang, and their co-authors. http://www.informatik.uni-bremen.de/agbs/jp/jp_papers_e.html
2. Chow, T.S.: Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.* **4**(3), 178–187 (1978).
3. ERTMS/ETCS SystemRequirements Specification, Chapter 3, Principles, volume Subset-026-3, Issue 3.4.0 (2015), available under <http://www.era.europa.eu/Document-Register/Pages/Set-2-System-Requirements-Specification.aspx>
4. Nancy Leveson. *SafeWare: System Safety and Computers*. Addison Wesley 1995.
5. Neil Storey. *Safety-critical Computer Systems*. Addison-Welly, 1996.
6. Alexandre Petrenko, Nina Yevtushenko: Adaptive Testing of Nondeterministic Systems with FSM. *HASE 2014*: 224-228.

Acknowledgements

I would like to express my gratitude to my friends and collaborators who inspired and contributed to the ideas presented in this talk.

Ana Cavalcanti, Anne E. Haxthausen, Wen-ling Huang, Christoph Hilken, Felix Hübner, John Fitzgerald, Peter Gorm Larsen, Till Mossakowski, Mohammad Reza Mousavi, Alexandre Petrenko, Uwe Schulze, Linh Hong Vu, Jim Woodcock, Cornelia Zahlten

The work presented here has been performed in the context of project Implementable Testing Theories for Cyber-physical systems (ITTCPS)
<http://www.informatik.uni-bremen.de/agbs/projects/ittcps/index.html>