Introduction to Model Checking and NuSMV

Cyrille Artho
KTH Royal Institute of Technology, Stockholm, Sweden
School of Electrical Engineering and Computer Science
Theoretical Computer Science
artho@kth.se

Outline

1. How to write (temporal) properties:
   (a) Linear Temporal Logic (LTL).
   (b) Computation Tree Logic (CTL).

2. How to use the NuSMV model checker, part 1.

Linear Temporal Logic (LTL)

1. **Propositions:** p, $\neg p$, $p \wedge q$, $p \vee q$ (also for subformulas)

2. **Temporal operators**

X	ne**x**t	$\bigcirc$
U	**u**ntil	
F	**f**inally	$\Diamond$
G	**g**lobally	$\Box$

Other operators exists but they are not used quite as frequently, and can be derived from the ones above.

Semantics: defined on *paths*

1. **Propositions:** p, $\neg p$, $p \wedge q$, $p \vee q$ (also for subformulas)
   Formula p is true if p holds in first state: $w \models p$ if $p \in w(0)$
   Negation: $w \models \neg p$ if $w \not\models p$; conjunct: $w \models p \wedge q$ if $w \models p$ and $w \models q$

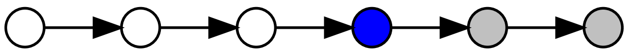
2. **Temporal operators**

X p	ne**x**t	$\bigcirc p$	$w \models \mathbf{X} p$ if $w_1 \models p$
p **U** q	**u**ntil		$\exists j \geq 0, w_j \models q \wedge \forall i, 0 \leq i < j, w_i \models p$
F p	**f**inally	$\Diamond p$	$\mathit{true}\, \mathbf{U}\, p$, equivalent to $\exists j \geq 0, w_j \models p$
G p	**g**lobally	$\Box p$	$\neg \mathbf{F} \neg p$, equivalent to $\forall j \geq 0, w_j \models p$

3. **Temporal operators can be nested.**
Furthermore, instead of just atomic propositions for p and q, we can also use other temporal formulas as subexpressions.

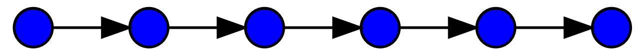
Examples

finally



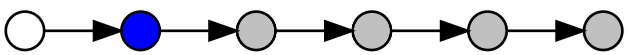
F p

globally



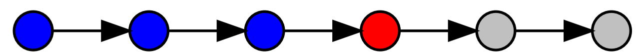
G p

next



X p

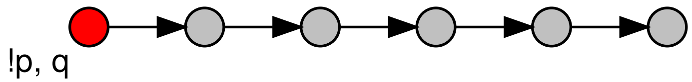
until



p **U** q

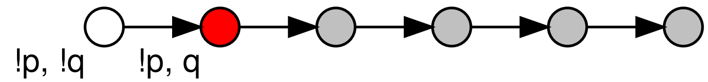
Until

true?



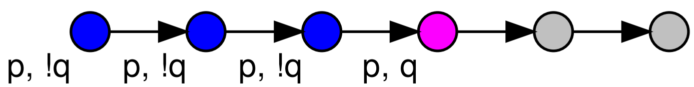
p **U** q

true?



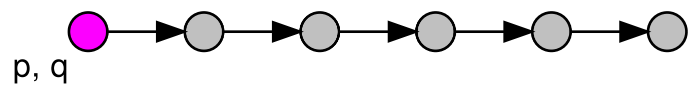
p **U** q

true?



p **U** q

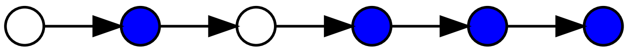
true?



p **U** q

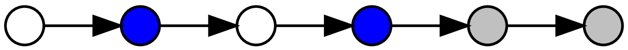
Nesting

true?



FG p

true?



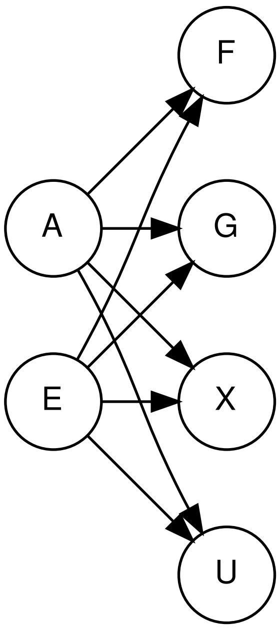
FG p

How about infinite traces?

Computation Tree Logic

1. **Propositions**

2. **Path quantifiers + temporal operators**



AF p: For all paths, p is eventually true.
AG p: For all paths, p is always true.
AX p: For all paths, p is true in the next state.
A[p **U** q]: For all paths, p holds until q holds.
E: There exists a path. . .

3. **Temporal operators always follow path quantifier**

Semantics: Defined on *transition systems*

1. **Propositions:** same as above

2. **Temporal operators** for model $M(S, \rightarrow, L)$

AX p	$((M,s) \models \mathbf{AX}p) \Leftrightarrow (\forall \langle s \rightarrow s_1 \rangle\, ((M,s_1) \models p))$
AG p	$((M,s) \models \mathbf{AG}p) \Leftrightarrow (\forall \langle s_1 \rightarrow s_2 \rightarrow \ldots \rangle (s = s_1) \forall i\, ((M,s_i) \models p))$
AF p	$((M,s) \models \mathbf{AF}p) \Leftrightarrow (\forall \langle s_1 \rightarrow s_2 \rightarrow \ldots \rangle (s = s_1) \exists i\, ((M,s_i) \models p))$
A [p **U** q]	$((M,s) \models \mathbf{A}[p\, \mathbf{U}\, q]) \Leftrightarrow \forall (\langle s_1 \rightarrow s_2 \rightarrow \ldots \rangle (s = s_1) \exists j$ $(((M,s_j) \models q) \wedge (\forall (i < j)(M,s_i) \models p)))$

E. . . defined analogously with existential path quantifier.

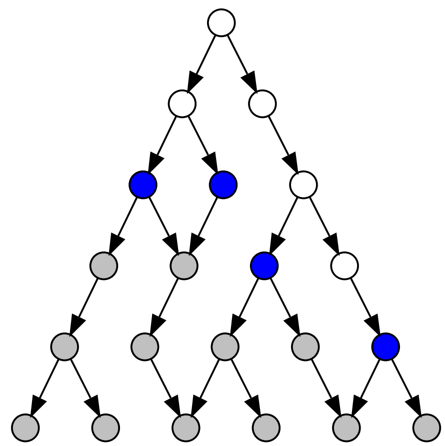
3. **Temporal operators canNOT be directly nested.**
However, instead of just atomic propositions for p and q,
we can also use other temporal formulas as subexpressions.

Computation Tree Logic (CTL): finally, globally

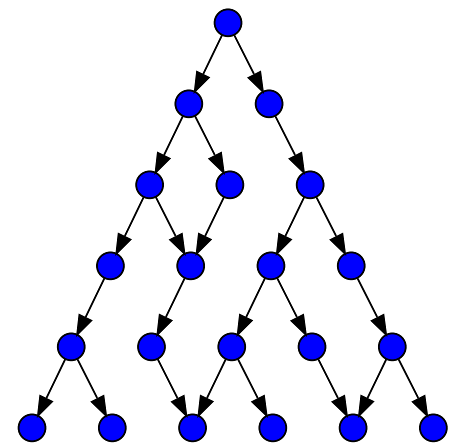
finally

globally

for **all** paths

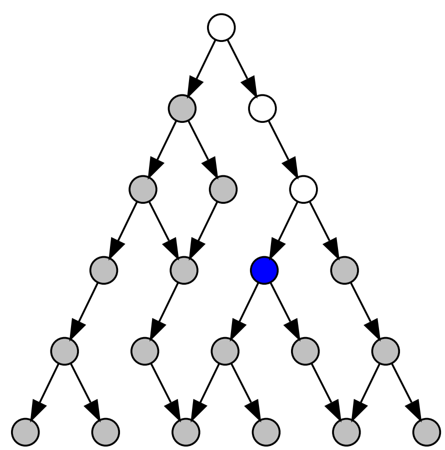


AF p

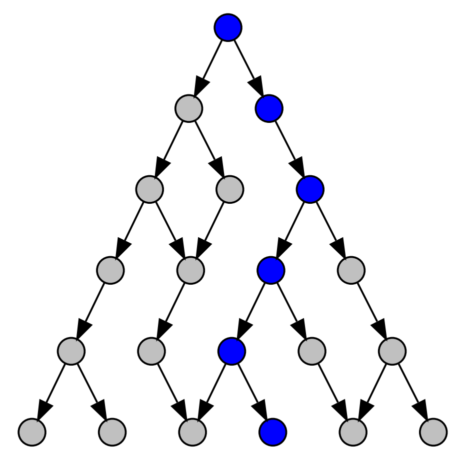


AG p

there **exists** a path

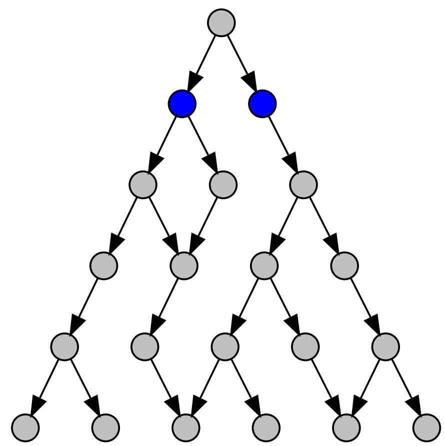
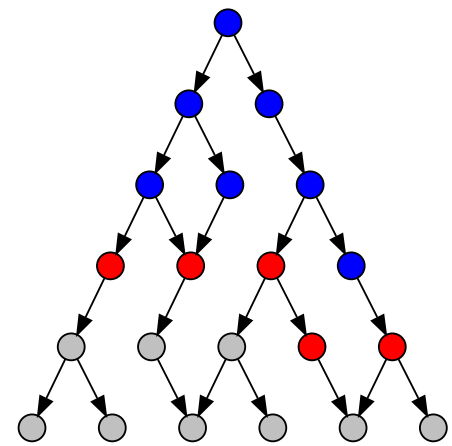
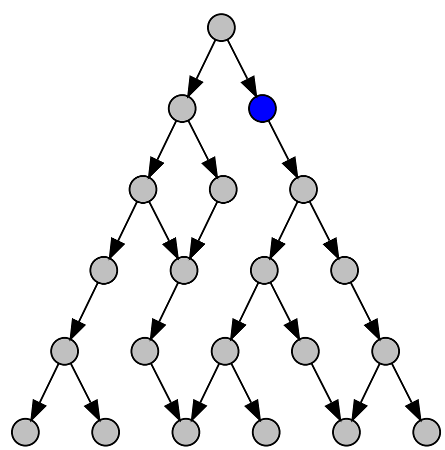
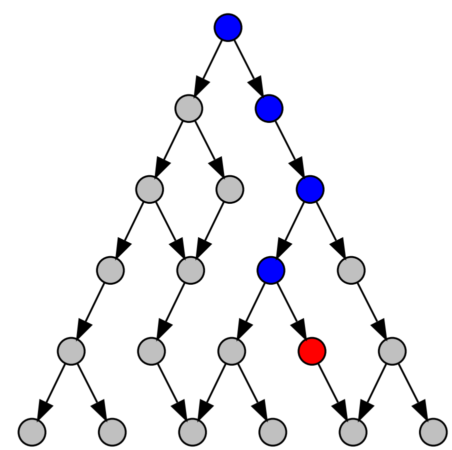


EF p



EG p

Computation Tree Logic (CTL): next, until

	next	until
for **all** paths	 **AX** p	 **A [** p **U** q **]**
there **exists** a path	 **EX** p	 **E [** p **U** q **]**

LTL $\neq$ CTL

1. **FG**(p)

2. **EX**(p)

- *Which logic can express the formulas above?*

- *What is the semantics of each formula?*

- *Where does the counterpart fail to express it?*

CTL*: Combines LTL and CTL.

SMV and NuSMV

- Symbolic Model Verifier (SMV):
  First practical symbolic model checker by Ken McMillan/CMU.
- Re-implementation NuSMV (open source) at IRST Trento, Italy.
- NuSMV is still being maintained and developed.
- Current version is 2.6.0 (used in this course).

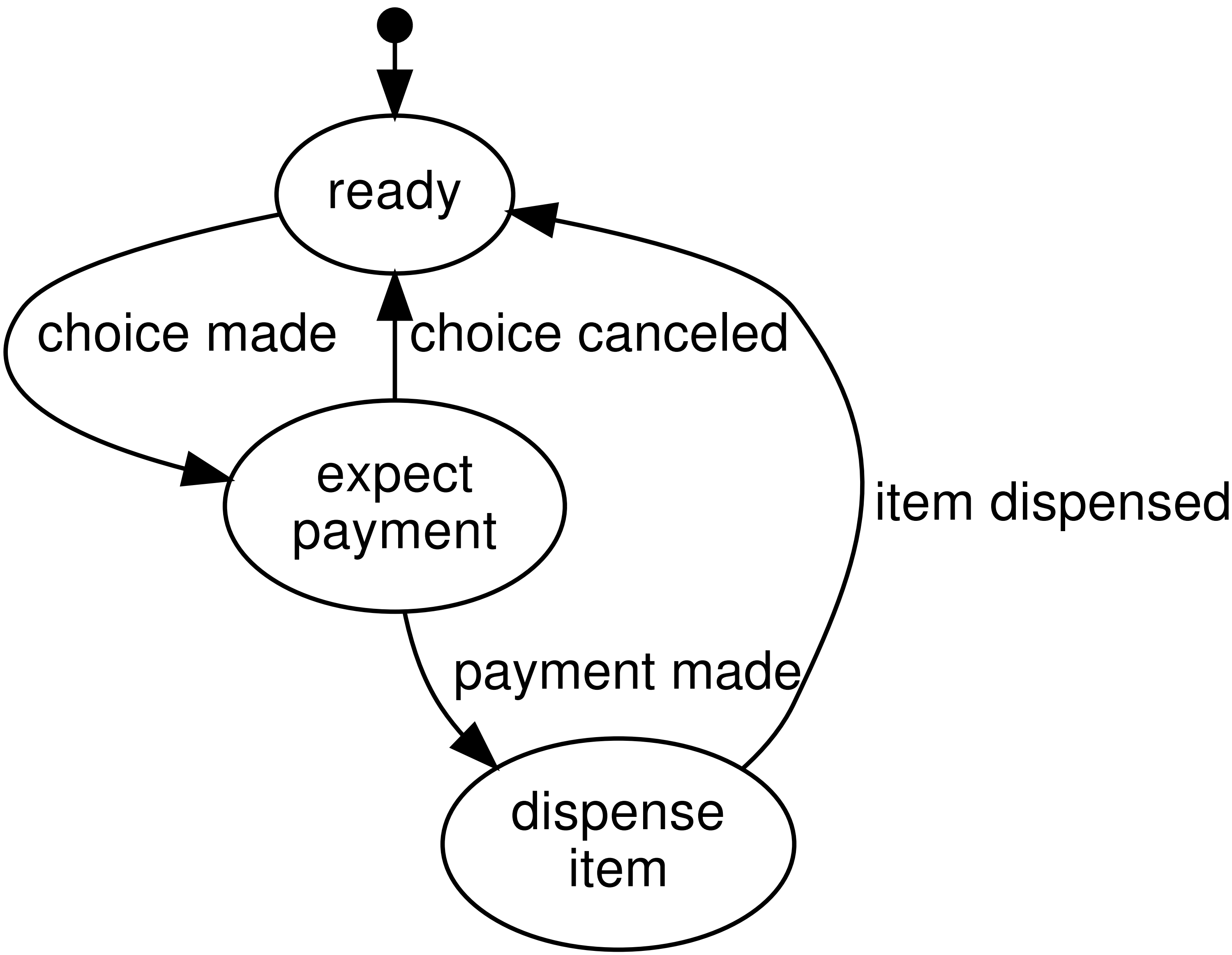
Usage of NuSMV in the real world

- As a back-end to other tools:
  - → NuSMV-PA: Safety analysis platform
  - → Back-end for Petri net model checking (another modeling approach).
  - → Test case generation.
- Case studies:
  - → Kerberos protocol.
  - → Web service composition.
  - → Railway interlocking control tables.

A critique of NuSMV

Pro	**Con**
open source	limited open tool set
mature	limited syntax
fast	no arrays of modules
well-defined semantics	focus on synchronous systems

Example: Vending machine



Details not modeled

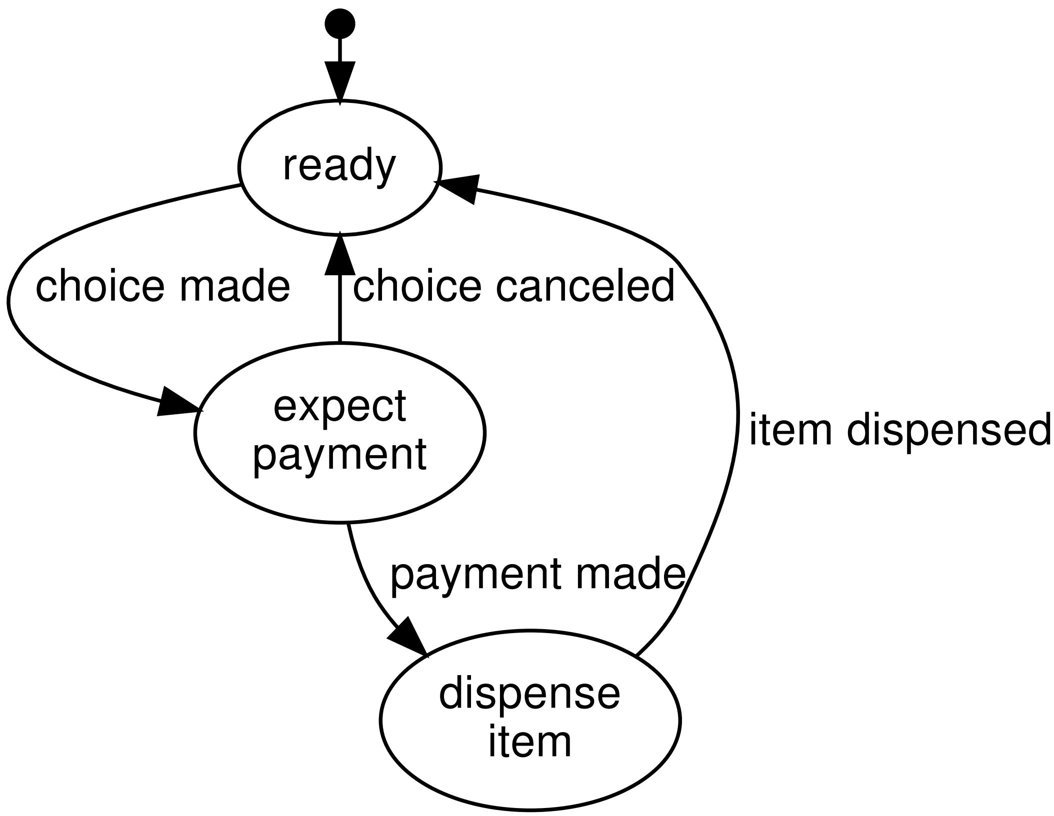
- How choice is made (how many choices) and canceled.

$\rightarrow$ Button, timeout, both?

- How payment is handled/accepted; price of goods.

- Item dispenser mechanism, time taken to dispense item.

Example: Vending machine—2



- Three states: ready, expect_payment, dispense_item.
- Two user inputs (non-deterministic):

1. Choice (of item); can be cancelled.
2. Payment (requires choice to be made first).

Vending machine in NuSMV

```
MODULE main
VAR
  choice, payment:  boolean;
  state:   { ready, expect_payment, dispense_item };
ASSIGN
  init (state) := ready;
  next (state) := case
    state = ready              & choice:  expect_payment;
    state = expect_payment & payment: dispense_item;
    state = expect_payment & !choice: ready;
    state = dispense_item:            ready;
  esac;
```

NuSMV syntax

- Module, variables, assignments.
- case (follows order of declaration); can have multiple outcomes.

Error message

```
file vending.smv: line 3: at token ",": syntax error
file vending.smv: line 3: Parser error
NuSMV terminated by a signal
```

- Cannot declare more than one variable per line!
- Try again...

Another error message

```
file vending.smv: line 14: case conditions are not exhaustive
```

- Recall the **`case`** block:

```
state = ready           & choice:  expect_payment;
state = expect_payment & payment: dispense_item;
state = expect_payment & !choice: ready;
state = dispense_item:             ready;
```

- State should remain the same by default: specify!

Complete `case` block

```
ASSIGN
  init (state) := ready;
  next (state) := case
    state = ready           & choice:  expect_payment;
    state = expect_payment & payment: dispense_item;
    state = expect_payment & !choice: ready;
    state = dispense_item:            ready;
    TRUE:                             state;
  esac;
```

- Transitions from ready and expect_payment depend on user choice.
- Cancellation is modeled as **`choice`** reverting to **`false`**.
- Transition from dispense_item back to ready is automatic.

Run NuSMV

- Nothing happens!
- We need properties...

```
LTLSPEC
  G(choice -> F state = dispense_item);
```

- „Every time I choose something, I eventually get it”.

Nice try!

```
-- specification
   G (choice ->  F state = dispense_item)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 1.1 <-
    choice = TRUE
    payment = FALSE
    state = ready
  -> State: 1.2 <-
    choice = FALSE
    state = expect_payment
  -> State: 1.3 <-
    choice = TRUE
    state = ready
```

OK, I'll pay…

```
LTLSPEC
   G(payment -> F state = dispense_item);
-- specification
   G (payment ->  F state = dispense_item)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 1.1 <-
    choice = FALSE
    payment = TRUE
    state = ready
  -> State: 1.2 <-
```

- State 1.2: no progress.
- Payment is accepted even when no choice has been made!

Accept payment only when choice is made

```
MODULE main
VAR
  choice:  boolean;
  payment: boolean;
  acc_payment: boolean;
  state:   { ready, expect_payment, dispense_item };
ASSIGN
  init (state) := ready;
  next (state) := case
    state = ready          & choice:   expect_payment;
    state = expect_payment & acc_payment: dispense_item;
    state = expect_payment & !choice: ready;
    state = dispense_item:            ready;
    TRUE:                             state;
  esac;
  init (acc_payment) := FALSE;
  next (acc_payment) := (state = expect_payment & payment);
```

Another problem?!

`G (acc_payment -> F state = dispense_item)` **is** false

```
-> State: 1.1 <-
  choice = FALSE
  payment = FALSE
  acc_payment = FALSE
  state = ready
-> State: 1.2 <-
  choice = TRUE
-> State: 1.3 <-
  choice = FALSE
  payment = TRUE
  state = expect_payment
-> State: 1.4 <-
  payment = FALSE
  acc_payment = TRUE
  state = ready
-- Loop starts here
-> State: 1.5 <-
  acc_payment = FALSE
-> State: 1.6 <-
```

State 1.3: Choice is made, next state = accept payment
State 1.4: accepting payment, but choice is canceled just now!

- Need a way to prevent this transition back to *ready*.
- Use stricter **`case`** condition!
- Lab exercise 1.

Extension of the vending machine

- Limited capacity of n items.
- Payment should not be accepted when no items available.
- Counting down items:

```
next(n_items) := case
  ...: n_items - 1;
  TRUE: n_items;
esac;
```

- Fails!

```
file vending3.smv: line 16:
  cannot assign value -1 to variable n_items
```

Counting without over- or underflow

- NuSMV recognized possible over- or underflow at compile time.

```
next(n_items) := case
    ... & n_items > 0: n_items - 1;
    TRUE: n_items;
esac;
```

- Underflow needs to be prevented in code.

Additional properties

1. Number of items should always be ≥ 0.
2. Payment should only be accepted if number of items > 0.
3. If an item is dispensed, the counter of items is always reduced by 1.

First lab exercise (part of assignment 1): Summary

1. Use NuSMV on **`vending1.smv`**.
   (a) Study error trace.
   (b) Refine transition (**`case`** condition).

2. Counting remaining items.
   (a) Add a counter **`n_items`** (see above).
   (b) Write LTL or CTL properties for the three properties above.
   (c) Ensure your model fulfills all properties.

Summary

Temporal logics

Linear temporal logic (LTL):

- No branching.
- Defined on paths.

Computational tree logic (CTL):

- Branching.
- Defined on transition systems.

Model checking

Enumerate all reachable states.

- Check if „bad” state reachable.
- Active research, many optimizations (such as BDDs).