

NuSMV Modelling Guide

1 What a NuSMV Model Describes	1
2 The Steps	1
2.1 Step 1 — Choose the Abstraction Level	1
2.2 Step 2 — Entities → Modules	1
2.3 Step 3 — State Variables & Types	1
2.4 Step 4 — Sentences → Constructs	2
2.5 Step 5 — ASSIGN vs. TRANS	2
2.6 Step 6 — Composition & Concurrency	3
2.7 Step 7 — Fairness	3
2.8 Step 8 — Properties & Sanity Checks	3
3 Worked Example	4
4 Checklist	5

1 What a NuSMV Model Describes

A model defines a Kripke structure $K = (S, I, \rightarrow, \lambda)$ (Def. 3.3):

Part	Meaning in NuSMV
S	finite set of states = every combination of variable values → every variable must be bounded
$I \neq \emptyset$	initial states — must be non-empty
$\rightarrow$	left-total : every state has ≥ 1 successor → no deadlocks, every path is infinite
λ	atomic propositions per state → Boolean expressions, best written as DEFINES

Most rules below exist to keep S small and to keep I and $\rightarrow$ well-formed.

2 The Steps

2.1 Step 1 — Choose the Abstraction Level

- Model **what the system does**, not the code. Leave out details no property needs (the elevator, §3.6, ignores doors and in-cabin buttons).
- A removed detail that decided between two actions becomes **non-determinism** (§3.2): if a/b depended on a **flag** you removed, the model now picks $\{a, b\}$.
- Too much detail → state-space explosion; too little → bugs get abstracted away. No formal rule — it takes experience (§3.5).

2.2 Step 2 — Entities → Modules

- One **MODULE** per real-world agent (button, elevator, counter); **main** creates the instances.
- Whatever one component **sees** of another becomes a **module parameter**, written **inst.var** (Fig. 3.10: `c : counter(inc.cnt = inc.mx)`).
- Per-instance constants (bounds, IDs) are parameters too: `increase(flag, mx), reqbutton(floor, elevfloor)`.
- Instances may refer to each other circularly; individual variables' **next** values may **not** depend on each other in a cycle.

2.3 Step 3 — State Variables & Types

The description has...	Type
on/off, yes/no	boolean
named phases or modes (best for control states)	enum $\{\text{idle, busy}\}$
counts, floors, positions	range $0..N$
bit-level behaviour	unsigned word $[n]$ / signed word $[n]$
one value per floor or process	array $0..n$ of T

- Bound everything.** “Count how many times...” → `nr_resets : 0..100`, with an arbitrary cap (Fig. 3.7).

- **As few variables as possible.** Anything computable from other variables goes in **DEFINE**, not **VAR** — a **DEFINE** adds no state (§3.3.2).
- **Fixed defaults.** When a variable doesn’t matter in the current state, reset it to a fixed value instead of keeping an old one — “a very important tip” for state-space size (§3.6).

2.4 Step 4 — Sentences → Constructs

This is the core of the conversion:

Natural language	NuSMV
“Initially ...”	<code>init(v) := ...;</code> (omit it → any value in range)
“When C, v becomes e”	<code>case row C : e;</code>
“may”, “can”, “at any time”, “either ... or”	set of values: <code>C : {a, b};</code>
“otherwise it stays the same”	last row <code>TRUE : v;</code>
“X takes priority over Y”	X’s row above Y’s (first match wins)
“these values change together / are related”	TRANS (Step 5)
“count how often v becomes z”	guard on <code>next(v) = z</code> (Fig. 3.7)
“A reacts to B’s state”	pass B’s state as a parameter
“the system is in situation ...” (observation)	DEFINE
“eventually” / “not forever” about the environment	FAIRNESS / COMPASSION (Step 7)
“must always / never / eventually” about the system	a property to check, not part of the model (Step 8)

case rows are **ordered**: a row only applies if every earlier row didn’t (Fig. 3.3 line 8 implicitly includes `!reset`). This silently creates priorities — the elevator always serves lower floors first (§3.6).

- NuSMV statically checks that every **case** covers all situations (left-totality) and that values stay in range.
- **Environment freedom must be written out.** Modules run in lockstep, so “a button can be pressed at any moment” has to be `!requested : {FALSE, TRUE};` (Fig. 3.30). Otherwise only one fixed timing is checked.

2.5 Step 5 — ASSIGN vs. TRANS

Default: ASSIGN (`init/next` per variable). NuSMV then guarantees values stay in range, every case is covered, and there are no circular dependencies.

INIT + TRANS (and no **ASSIGN** in that module) when updates depend on each other, or a relation is easier to write than a function:

```
TRANS next(x) + 2*next(y) + 3*next(z)
      = (x + 2*y + 3*z) mod 7
```

Typical shape — `|`-joined “guard & effect” options (Fig. 3.8):

```
INIT state = zero
TRANS (reset & next(state) = zero) |
      (!reset & state = zero & next(state) = one) |
      ...
```

With **TRANS**, two checks become **your** job (§3.3.1): (1) **INIT** is satisfiable, (2) **TRANS** is left-total. If either fails, every result is meaningless:

Broken model	NuSMV says
<code>x : 0..3 with INIT x = 5</code>	<code>G x = 2</code> is true
<code>TRANS next(x) = x + 1</code>	<code>AG x < 3</code> is true , although <code>x = 3</code> is reachable

Always run `NuSMV -ctt` (it reports “transition relation is not total ... deadlock state `x = 3`”) and never ignore the warning “Fair states set ... is empty”.

(Not in book.) **TRANS** is a constraint, not an assignment: a variable an option doesn't mention can take **any** next value. Pin unchanged variables with `next(y) = y`.

2.6 Step 6 — Composition & Concurrency

- **Synchronous**: all instances step together on every tick. Nothing is missed or counted twice, and state spaces stay small — elevator with 3 floors: 88 states in NuSMV vs. 7,957 in Promela (Table 3.4).
- **Interleaving** (asynchronous **process** is deprecated): add a control variable such as `run : {a, b}` chosen non-deterministically. Each module only changes when selected, else `TRUE : v`.
- **Sequences** (“take request, then set direction, then move”) → chain `next` dependencies over successive steps (elevator: `reqfloor` → `status` → `curfloor`).
- **Interfaces via DEFINE** (`lastState`, `mx_reached`) let you swap implementations (`state` enum vs. `val : 0..2`) without changing other modules (Fig. 3.11).

2.7 Step 7 — Fairness

When: a liveness property fails, but **only** on an unrealistic infinite path where an enabled step never happens — the counter resets forever, a process is never scheduled, a user prints forever.

	Weak (justice)	Strong (compassion)
Meaning	continuously enabled → eventually taken	enabled infinitely often → taken infinitely often
NuSMV	<code>FAIRNESS p / JUSTICE p</code> = “p infinitely often”; for transition <i>t</i> : p = not enabled or taken	<code>COMPASSION(p, q)</code> with p = enabled, q = taken
LTL	$G F(\neg en_t \vee tk_t)$	$G F en_t \rightarrow G F tk_t$
Use when	the step stays enabled the whole time it waits	the step's enabledness flickers on and off

Flickering example (Fig. 4.5): `reset` toggles every step, so `zero` → `one` is only enabled every other step. Weak fairness doesn't force it; strong fairness does. In practice it's usually one simple predicate: `FAIRNESS !(ctr.state = zero)`; (Ex. 4.6).

- Fairness **can't be a CTL formula** (§4.8); NuSMV instead restricts A/E to fair paths. In LTL it can also be inlined: `LTLSPEC (G F p) -> phi` (§4.6).
- Fairness is an **assumption about the environment or scheduler**. Never use it to hide a flaw in the part you are designing (see the worked example).
- Don't add fairness for “may” behaviour: a user who **may** request can legitimately stay idle forever.
- `next()` is **not allowed** inside `FAIRNESS` / `JUSTICE` / `COMPASSION` (parse error).

(Not in book.) Fairness can **hide safety violations** in CTLSPEC/LTLSPEC. Test: `x` goes `0` → `{0, 1}`, `1` → `{0, 2}`, and `x = 2` is a trap, with `FAIRNESS x = 0`. Both `AG x != 2` and `G x != 2` came out **true**, but `INVARSPEC x != 2` came out **false**. `INVARSPEC` ignores fairness → use it for invariants.

2.8 Step 8 — Properties & Sanity Checks

- Write atomic propositions as **DEFINES while modelling**: “we anticipate the properties that we wish to express” (§3.5).
- **Safety** (“nothing bad happens”): finite counterexample. Bounded liveness (“within 10 min”) and deadlock freedom count as safety.
- **Liveness** (“something good eventually happens”): infinite counterexample; often needs fairness.
- **LTL (LTLSPEC)**: all paths. `F G p` is LTL-only. NuSMV has no weak until: `a W b = (a U b) | G a`.
- **CTL (CTLSPEC / SPEC)**: **possibility**, e.g. `EF`, `AG EF start` — CTL-only.

- **Sanity:** add specs that should hold for a real reason (EF goal), make variants that should fail and check they do, and run NuSMV -r -ctt model.smv (-r reachable-state count, -ctt totality).

3 Worked Example

Two users share a printer. Each user is idle, waiting or printing. An idle user may request at any time. A waiting user starts printing once the arbiter grants access. A printing user eventually finishes. The arbiter grants access to at most one user, and alternates when both are waiting. Requirements: never both printing; a waiting user eventually prints; it is always possible to get back to both users idle.

Sentence	Becomes
users, arbiter	modules
“may request at any time”	{idle, waiting}
“eventually finishes”	{printing, idle} + fairness
“alternates when both are waiting”	turn variable
requirements	INVARSPEC / LTLSPEC / CTLSPEC

User — the environment: free to request, eventually finishes.

```

MODULE user(granted)
VAR
  st : {idle, waiting, printing};
ASSIGN
  init(st) := idle;
  next(st) := case
    st = idle           : {idle, waiting};
    st = waiting & granted : printing;
    st = waiting        : waiting;
    st = printing       : {printing, idle};
  esac;
DEFINE
  wants  := st = waiting;
  using  := st = printing;
  is_idle := st = idle;
FAIRNESS
  !using -- never prints forever

```

Arbiter — the part being designed: one owner at a time, alternates on ties.

```

MODULE arbiter(w1, w2, idle1, idle2)
VAR
  owner : {none, one, two};
  turn  : {one, two};
ASSIGN
  init(owner) := none;
  init(turn)  := one;
  next(owner) := case
    owner = one & idle1 : none;
    owner = two & idle2 : none;
    owner != none       : owner;
    w1 & w2              : turn;  -- tie-break
    w1                  : one;
    w2                  : two;
    TRUE                : none;
  esac;
  next(turn) := case
    owner = none & w1 & w2 & turn = one : two;
    owner = none & w1 & w2 & turn = two : one;
    TRUE                                : turn;
  esac;

```

Main — wiring through parameters, plus the specs.

```

MODULE main
VAR
  u1 : user(arb.owner = one);
  u2 : user(arb.owner = two);
  arb : arbiter(u1.wants, u2.wants,
               u1.is_idle, u2.is_idle);

INVARSPEC !(u1.using & u2.using)
LTLSPEC   G (u1.wants -> F u1.using)
LTLSPEC   G (u2.wants -> F u2.using)
CTLSPEC   AG EF (u1.is_idle & u2.is_idle)
CTLSPEC   EF u1.using                    -- sanity

```

Variant	NuSMV result
As written	all 5 specs true ; transition relation total; 32 reachable states
Without FAIRNESS !using	both liveness specs false : one user prints forever → unrealistic environment be- haviour, so fairness is the right fix
w1 & w2 : one instead of turn	u2's liveness false even with fairness → a real design bug (same priority bias as the elevator's case order); fix the model, not with fairness

4 Checklist

1. Choose the abstraction; removed details become non-deterministic choices.
2. Entities → modules, interactions → parameters, `main` connects them.
3. Few, bounded variables, with fixed defaults when a value doesn't matter.
4. Translate sentence by sentence; watch the `case` row order.
5. `ASSIGN` by default. `TRANS` only for related updates — then run `-ctt`.

6. Environment freedom → explicit `{...}` choices.
7. Observations → `DEFINES` (atomic propositions).
8. Requirements → `INVARSPEC` / `LTLSPEC` / `CTLSPEC`, plus sanity specs that should hold or fail.
9. Liveness fails only on an unrealistic path → `FAIRNESS` (step stays enabled) or `COMPASSION` (enabledness flickers). Never use fairness to excuse a design flaw.