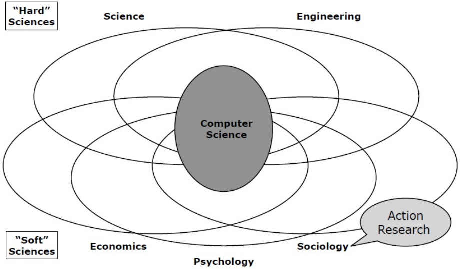
Model Checking



Outline

Critical Software

Single programs

- Operating systems
- Crypto routines
- Financial systems
- Medical devices
- Flight control systems
- Power plants
- Home security
- …

Programming languages

- Static type systems
- Data abstraction and modularity
- Security controls
- Compiler correctness

Logic

+ Reasoning about
individual programs

+ Reasoning about
whole programming
languages

SOFTWARE FOUNDATIONS

Building Reliable Software

- Suppose you work at (or run) a software company.
- Suppose, like Frege, you've sunk 30+ person-years into developing the "next big thing":
  - Boeing Dreamliner2 flight controller
  - Autonomous vehicle control software for Nissan
  - Gene therapy DNA tailoring algorithms
  - Super-efficient green-energy power grid controller
- Suppose, like Frege, your company has invested a lot of material resources that are also at stake.
- How do you avoid getting a letter like the one from Russell?

Or, worse yet, *not* getting the letter,
with disastrous consequences down the road?

Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - “lint” tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound type systems
  - Formal verification

Less “formal”: Lightweight, inexpensive techniques (that may miss problems)

This isn’t an either/or tradeoff… a spectrum of methods is needed!

Even the most “formal” argument can still have holes:

- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth: *“Beware of bugs in the above code; I have only proved it correct, not tried it.”*

More “formal”: eliminate *with certainty* as many problems as possible.

How to Build Reliable Software

- Your Project Proposal

Definition

- Create a formal model of some system of interest
  - Hardware
  - Communication Protocol
  - Software, esp. concurrent software
- Describe formally a specification that we desire to model to satisfy
- Check the model satisfies the specification
  - Theorem Proving (usually interactive but not necessarrily)
  - Model Checking

Example of Specification: SpaceWire Protocol (European Space Agengy Standard)

8.5.2.2 ErrorReset

a. The *ErrorReset* state shall be entered after a system reset, after link operation is terminated for any reason or if there is an error during link initialization.

b. In the *ErrorReset* state the Transmitter and Receiver shall all be reset.

c. When the reset signal is de-asserted the *ErrorReset* state shall be left unconditionally after a delay of 6,4 µs (nominal) and the state machine shall move to the *ErrorWait* state.

d. Whenever the reset signal is asserted the state machine shall move immediately to the *ErrorReset* state and remain there until the reset signal is de-asserted.

Interpreatation $\models$ Formula

The relationship between interpretations M and formulas ϕ:

$$M \models \phi$$

We say M **models** ϕ.

Questions we can ask:

1. For a fixed ϕ, is $M \models \phi$ true for all M?
   - Validity of ϕ
   - This can be done via proof in a theorem prover e.g. Isabelle.
2. For a fixed ϕ, is $M \models \phi$ true for some M?
   - Satisfiability
3. For a fixed (class of) M, what ϕs make $M \models \phi$ true?
   - "Theory discovery"/"Learning from Data"/"Generalisation"
   - Not in this course
4. For a fixed M and P, is it the case that $M \models \phi$?
   - Model Checking

Model Checking - Definition

At a high level, many tasks can be rephrased as model checking.

"Interpretations" M	$\models$	"Formulas" ϕ	Task
sequences of tokens	$\models$	grammars	parsing
database tables	$\models$	SQL queries	query execution
email texts	$\models$	spam rules	spam detection
sequences of letters	$\models$	dictionary	spellchecking
audio data	$\models$	acoustic/lang. model	speech recognition
finite state machines	$\models$	temporal logic	specification checking

Details differ widely, but question of "is this data consistent with this statement? (and to what degree?)" is extremely common.

Historically, "Model Checking" usually refers to the last one.

Model Checking - Models

A model of some system has:

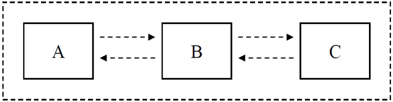
- A finite set of **states**
- A subset of states considered as the **initial states**
- A **transition relation** which, given a state, describes all states that can be reached "in one time step".

Good for

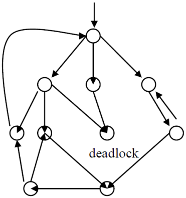
- Software, sequential and concurrent
- Digital hardware
- Communication protocols

Refinements of this setup can handle: **Infinite state spaces**, **Continuous state spaces**, **Continuous time**, **Probabilistic Transitions**. Good for hybrid (*i.e.*, discrete and continuous) and control systems.

Model Checking - Models



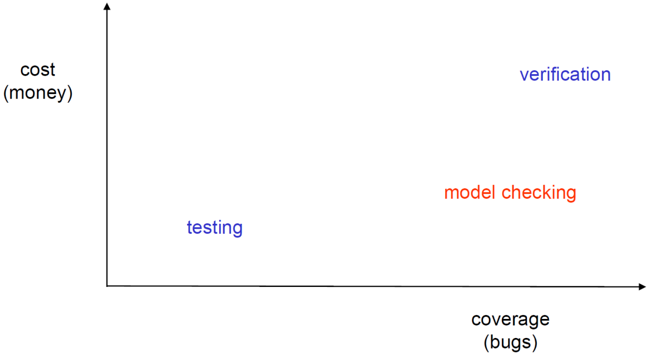
Each component is modeled by a FSM.



- Model Checking (MC) is
  - check whether a program satisfies a property by exploring its state space
  - systematic state-space exploration = exhaustive testing
  - "check whether the system satisfies a temporal-logic formula"
- Simple yet effective technique for finding bugs in high-level hardware and software designs
- Once thoroughly checked, models can be compiled and used as the core of the implementation

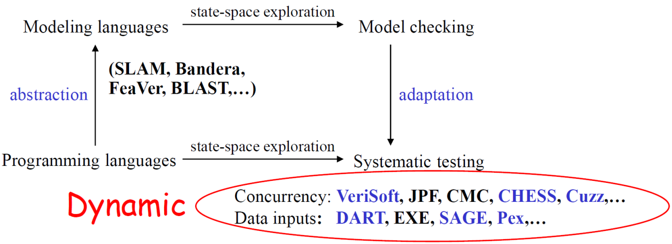
Insight: Model Checking is Super Testing

- Simple yet effective technique for finding bugs



Software Model Checking

- How to apply model checking to analyze **software**?
  - "Real" programming languages (e.g., C, C++, Java),
  - "Real" size (e.g., 100,000's lines of code)
- Two main approaches to software model checking:



Model Checking - Specification

We are interested in specifying behaviours of systems over time.

- Use **Temporal Logic**

Specifications are built from:

1. Primitive properties of individual states
   e.g., "is on", "is off", "is active", "is reading";
2. propositional connectives $\wedge, \vee, \neg, \rightarrow$;
3. and **temporal** connectives: *e.g.*,
   At **all times**, the system is not simultaneously *reading* and *writing*.
   If a *request* signal is asserted **at some time**, a corresponding *grant* signal will be asserted **within 10 time units**.

The exact set of temporal connectives differs across temporal logics.
Logics can differ in how they treat time:

- **Linear time** vs. **Branching time**

These differ in reasoning about *non-determinism*.

Non-Determinism

In general, system descriptions are *non-deterministic.*

A system is *non-deterministic* when, from some state there are **multiple** alternative next states to which the system could transition.

Non-determinism is good for:

- Modelling alternative inputs to the system from its environment (*External non-determinism*)
- Under-specifying the model, allowing it to capture many possible system implementations (*Internal non-determinism*)

Linear vs. Branching Time

- **Linear Time**
  - Considers paths (sequences of states)
  - If system is non-deterministic, many paths for each initial state
  - Questions of the form:
    - For all paths, does some path property hold?
    - Does there exist a path such that some path property holds?
- **Branching Time**
  - Considers tree of possible future states from each initial state
  - If system is non-deterministic from some state, tree forks
  - Questions can become more complex, *e.g.*,
    - For all states reachable from an initial state, does there exist an onwards path to a state satisfying some property?
  - Most-basic branching-time logic (CTL) is complementary to most-basic linear-time logic (LTL)
  - Richer branching-time logic (CTL*) incorporates CTL and LTL.

LTL - Syntax

LTL = Linear(-time) Temporal Logic

Assume some set *Atom* of atomic propositions

Syntax of LTL formulas ϕ:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \rightarrow \phi \mid \mathbf{X}\phi \mid \mathbf{F}\phi \mid \mathbf{G}\phi \mid \phi\mathbf{U}\phi$$

where $p \in Atom$.

Pronunciation:

- $\mathbf{X}\phi$ — neXt ϕ
- $\mathbf{F}\phi$ — Future ϕ
- $\mathbf{G}\phi$ — Globally ϕ
- $\phi\mathbf{U}\psi$ — ϕ Until ψ

Other common connectives: **W** (weak until), **R** (release).

Precedence high-to-low: $(\mathbf{X}, \mathbf{F}, \mathbf{G}, \neg)$, $(\mathbf{U})$, $(\wedge, \vee)$, $\rightarrow$.

- E.g. Write $\mathbf{F}p \wedge \mathbf{G}q \rightarrow p\,\mathbf{U}\,r$ instead of $((\mathbf{F}p) \wedge (\mathbf{G}q)) \rightarrow (p\,\mathbf{U}\,r)$.

LTL - Informal Semantics

LTL formulas are evaluated at a position i along a path π through the system (a path is a sequence of states connected by transitions)

- An atomic p holds if p is true the state at position i.
- The propositional connectives $\neg, \wedge, \vee, \rightarrow$ have their usual meanings.
- Meaning of LTL connectives:
  - $\mathbf{X}\phi$ holds if ϕ holds at the next position;
  - $\mathbf{F}\phi$ holds if there exists a future position where ϕ holds;
  - $\mathbf{G}\phi$ holds if, for all future positions, ϕ holds;
  - $\phi\mathbf{U}\psi$ holds if there is a future position where ψ holds, and ϕ holds for all positions prior to that.
  - $\phi\mathbf{R}\psi$ holds if there is a future position where ϕ becomes true, and ψ holds for all positions prior to and including that i.e. ϕ 'releases' ψ.
    - It is equivalent to $\neg(\neg\phi\mathbf{U}\neg\psi)$.
    - Thus $\mathbf{R}$ is the dual of $\mathbf{U}$.

A Taste of LTL - Examples

1. $\mathbf{G}\ invariant$
   invariant is true for all future positions
2. $\mathbf{G}\ \neg(read \wedge write)$
   In all future positions, it is not the case that *read* and *write*
3. $\mathbf{G}(request \rightarrow \mathbf{F} grant)$
   At every position in the future, a *request* implies that there exists a future point where *grant* holds.
4. $\mathbf{G}(request \rightarrow (request\ \mathbf{U}\ grant))$
   At every position in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.
5. $\mathbf{G}\ \mathbf{F}\ enabled$
   In all future positions, there is a future position where *enabled* holds.
6. $\mathbf{F}\ \mathbf{G}\ enabled$
   There is a future position, from which all future positions have *enabled* holding.

LTL - Semantics: Formally

We want to define formally the satisfaction relation: $\sigma \models \phi$.

What kind of object is σ ?

An infinite trace of **sets of atomic propositions**:

$$\sigma \in (2^P)^\omega.$$

That is,

$$\sigma = \sigma_0, \sigma_1, \sigma_2, \cdots$$

where $\sigma_i \subseteq P$ for all i. P is the set of all atomic propositions.

Let $P = \{p, q\}$. Examples of traces:

$$\begin{array}{rcl} \sigma & = & \{p\}, \{q\}, \{p\}, \{q\}, \{p\}, \ldots \\ \rho & = & \{p\}, \{p\}, \{p\}, \{p\}, \{p\}, \ldots \\ \tau & = & \{p\}, \{q\}, \{p, q\}, \{\}, \{p, q\}, \ldots \\ \cdots & & \end{array}$$

LTL - Semantics: Formally

Let

$$\sigma = \sigma_0, \sigma_1, \sigma_2, \cdots$$

Notation: $\sigma[i..] = \sigma_i, \sigma_{i+1}, \sigma_{i+2}, \cdots$

Satisfaction relation defined recursively on the syntax of a formula:

$\sigma \models p$	iff	$p \in \sigma_0$	p holds at the first (current) step
$\sigma \models \phi_1 \wedge \phi_2$	iff	$\sigma \models \phi_1$ and $\sigma \models \phi_2$	
$\sigma \models \neg\phi$	iff	$\sigma \not\models \phi$	
$\sigma \models \mathsf{G}\phi$	iff	$\forall i = 0, 1, \ldots : \sigma[i..] \models \phi$	ϕ holds for every suffix of σ
$\sigma \models \mathsf{F}\phi$	iff	$\exists i = 0, 1, \ldots : \sigma[i..] \models \phi$	ϕ holds for some suffix of σ
$\sigma \models \mathsf{X}\phi$	iff	$\sigma[1..] \models \phi$	ϕ holds for the suffix starting at the next step
$\sigma \models \phi_1 \,\mathsf{U}\, \phi_2$	iff	$\exists i = 0, 1, \ldots : \sigma[i..] \models \phi_2 \wedge$ $\forall 0 \leq j < i : \sigma[j..] \models \phi_1$	ϕ_2 holds for some suffix of σ and ϕ_1 holds for all previous suffixes

LTL - Formal Semantics: Transition Systems and Paths

Definition (Transition System)

A *transition system* (or model) $\mathcal{M} = \langle S, \rightarrow, L \rangle$ consists of:

$$\begin{array}{cl} S & \text{a finite set of states} \\ \rightarrow \subseteq S \times S & \text{transition relation} \\ L : S \rightarrow \mathcal{P}(Atom) & \text{a labelling function} \end{array}$$

such that $\forall s_1 \in S.\ \exists s_2 \in S.\ s_1 \rightarrow s_2$

Note: $Atom$ is a fixed set of atomic propositions, $\mathcal{P}(Atom)$ is the powerset of $Atom$.

Thus, $L(s)$ is just the set of atomic propositions that is true in state s.

Definition (Path)

A *path* π in a transition system $\mathcal{M} = \langle S, \rightarrow, L \rangle$ is an infinite sequence of states $s_0, s_1, \ldots$ such that $\forall i \geq 0.\ s_i \rightarrow s_{i+1}$.

Paths are written as: $\pi = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$

Execution Traces of a State Machine

A **run** of a Mealy machine $(I, O, S, s_0, \delta, \lambda)$ is a (finite or infinite) sequence of states / transitions:

$$s_0 \xrightarrow{x_0/y_0} s_1 \xrightarrow{x_1/y_1} s_2 \xrightarrow{x_2/y_2} s_3 \cdots$$

such that

- $\forall i : x_i \in I, y_i \in O$
- $\forall i : s_{i+1} = \delta(s_i, x_i)$
- $\forall i : y_i = \lambda(s_i, x_i)$

The observable I/O behavior (**trace**) corresponding to the above run is

$$\{x_0, y_0\} \longrightarrow \{x_1, y_1\} \longrightarrow \{x_2, y_2\} \longrightarrow \cdots$$

Here we assume that only I/O are observable. We could also define traces that expose the internal state of the machine. E.g., we may want to state the requirement that a certain register never has a certain value.

LTL - Formal Semantics: Alternative Satisfaction By Path

Alternatively, we can define $\pi \models \phi$ using the notion of ith suffix $\pi^i = s_i \rightarrow s_{i+1} \rightarrow \ldots$ of a path $\pi = s_0 \rightarrow s_1 \rightarrow \ldots$.

For example, the alternative definition of satisfaction for G would be:

$$\pi \models \mathbf{G}\,\phi \qquad \text{iff} \qquad \forall j \geq 0.\ \pi^j \models \phi$$

instead of

$$\pi \models^0 \mathbf{G}\,\phi \qquad \text{iff} \qquad \forall j \geq 0.\ \pi \models^j \phi$$

Satisfaction in terms of $\models$ for the other connectives is left as an exercise.

- $\pi \models^i \phi$ is better for understanding, and needed for past-time operators.
- $\pi \models \phi$ is needed for the semantics of branching-time logics, like CTL.

LTL Semantics: Satisfaction by a Model

For a model $\mathcal{M}$, we write

$$\mathcal{M}, s \models \phi$$

if, for **every** execution path $\pi \in \mathcal{M}$ starting at state s, we have

$$\pi \models^0 \phi$$

A Taste of LTL - Examples

1. $\pi \models^i \mathbf{G}\, invariant$

   invariant is true for all future positions

   $\forall j \geq i.\ \pi \models^j invariant$

   $\forall j \geq i.\ invariant \in L(s_j)$

2. $\pi \models^i \mathbf{G}\, \neg(read \wedge write)$

   In all future positions, it is not the case that *read* and *write*

   $\forall j \geq i.\ read \notin L(s_j) \vee write \notin L(s_j)$

3. $\pi \models^i \mathbf{G}(request \rightarrow \mathbf{F} grant)$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds.

   $\forall j \geq i.\ request \in L(s_j)$ implies $\exists k \geq j.\ grant \in L(s_k)$.

4. $\pi \models^i \mathbf{G}(request \rightarrow (request\, \mathbf{U}\, grant))$

   At every position in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.

   $\forall j \geq i.\ request \in L(s_j)$ implies

   $\exists k \geq j.\ grant \in L(s_k)$ and $\forall l \in \{j, k-1\}.\ request \in L(s_l)$.

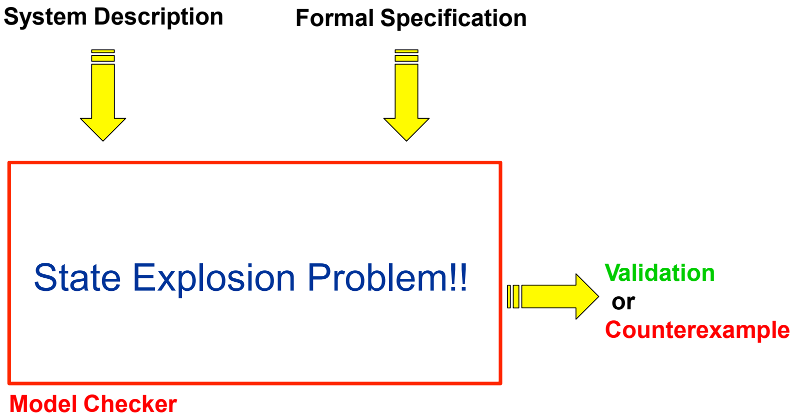
Model Checking Problem

Let ***M*** be a model, i.e., a **state-transition graph**.

Let ***f*** be the **property** in temporal logic.

Find all states ***s*** such that ***M*** has property ***f*** at state ***s***.

Model Checker Architecture



The State Explosion Problem

System Description

State Transition Graph

Combinatorial explosion of system states renders explicit model construction infeasible.

Exponential Growth of …
… global state space in number of concurrent components.
… memory states in memory size.

Feasibility of model checking inherently tied to handling state explosion.

Combating State Explosion

- **Binary Decision Diagrams** can be used to represent state transition systems more efficiently.
  ⇉**Symbolic Model Checking 1992**

- **Semantic techniques** for alleviating state explosion:
  - Partial Order Reduction.
  - Abstraction.
  - Compositional reasoning.
  - Symmetry.
  - Cone of influence reduction.
  - Semantic minimization.

Mochel Checking and Testing (papers and tools)

- Software Verification: Testing vs. Model Checking
  - https://www.sosy-lab.org/research/test-study/
- Sofware Testing via Model Checking
  - https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/main-24.pdf
- Reference Papers posted in BB (week 4)
- Spin: http://spinroot.com/spin/whatispin.html
- NuSMV: https://nusmv.fbk.eu/

NuSMV

NuSMV is a symbolic model checker developed by ITC-IRST and UniTN with the collaboration of CMU and UniGE.

`http://nusmv.fbk.eu/`

The NuSMV project aims at the development of a state-of-the-art model checker that:

- is robust, open and customizable;
- can be applied in technology transfer projects;
- can be used as research tool in different domains.

NuSMV is OpenSource

- developed by a distributed community, "Free Software" license

NuSMV

NuSMV provides:

1. A language for describing finite state models of systems
    - Reasonably expressive
    - Allows for modular construction of models
2. Model checking algorithms for checking specifications written in LTL and CTL (and some other logics) against finite state machines.

A first SMV program

```
MODULE main
  VAR
    b0 : boolean
  ASSIGN
    init(b0) := FALSE;
    next(b0) := !b0;
```

An SMV program consists of:

- Declarations of state variables (`b0` in the example); these determine the state space of the model.
- Assignments that constrain the valid initial states (`init(b0) := FALSE`).
- Assignments that constrain the transition relation (`next(b0) := !b0`).

Declaring state variables

SMV data types include:

boolean:

```
x : boolean;
```

enumeration:

```
st : {ready, busy, waiting, stopped};
```

bounded integers (intervals):

```
n : 1..8;
```

arrays and bit-vectors

```
arr : array 0..3 of {red, green, blue};
bv  : signed word[8];
```

Assignments

initialisation:

```
ASSIGN
init(x) := expression ;
```

progression:

```
ASSIGN
next(x) := expression ;
```

immediate:

```
ASSIGN
y := expression ;
```

or

```
DEFINE
y := expression ;
```

Assignments

- If no **init()** assignment is specified for a variable, then it is initialised non-deterministically;
- If no **next()** assignment is specified, then it evolves nondeterministically. i.e. it is unconstrained.
  - Unconstrained variables can be used to model nondeterministic inputs to the system.
- Immediate assignments constrain the current value of a variable in terms of the current values of other variables.
  - Immediate assignments can be used to model outputs of the system.

Expressions

$$
\begin{array}{rcll}
\mathit{expr} & ::= & \text{atom} & \text{symbolic constant} \\
 & | & \text{number} & \text{numeric constant} \\
 & | & \text{id} & \text{variable identifier} \\
 & | & \texttt{!}\,\mathit{expr} & \text{logical not} \\
 & | & \mathit{expr} \bowtie \mathit{expr} & \text{binary operation} \\
 & | & \mathit{expr}\texttt{[}\mathit{expr}\texttt{]} & \text{array lookup} \\
 & | & \texttt{next}(\mathit{expr}) & \text{next value} \\
 & | & \mathit{case\_expr} & \\
 & | & \mathit{set\_expr} &
\end{array}
$$

where $\bowtie \in \{\texttt{\&}, \texttt{|}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{=}, \texttt{!=}, \texttt{<}, \texttt{<=}, \ldots\}$

Case Expression

$$
\begin{array}{l}
\textit{case\_expr} ::= \\
\quad \texttt{case} \\
\qquad \textit{expr}_{a1} \;\; \texttt{:} \;\; \textit{expr}_{b1}\,\texttt{;} \\
\qquad \ldots \\
\qquad \textit{expr}_{an} \;\; \texttt{:} \;\; \textit{expr}_{bn}\,\texttt{;} \\
\quad \texttt{esac}
\end{array}
$$

- Guards are evaluated sequentially.
- The first true guard determines the resulting value

Set expressions

Expressions in SMV do not necessarily evaluate to one value.

- In general, they can represent a set of possible values.
  `init(var) := {a,b,c} union {x,y,z} ;`
- destination (lhs) can take any value in the set represented by the set expression (rhs)
- constant `c` is a syntactic abbreviation for singleton `{c}`

LTL Specifications

- LTL properties are specified with the keyword `LTLSPEC`:
  `LTLSPEC <ltl_expression> ;`
- `<ltl_expression>` can contain the temporal operators:
  `X_ F_ G_ _U_`
- E.g. condition `out = 0` holds until `reset` becomes false:
  `LTLSPEC (out = 0) U (!reset)`

ATM Example

```
MODULE main
VAR
  state: {welcome, enterPin, tryAgain, askAmount,
          thanksGoodbye, sorry};
  action: {cardIn, correctPin, wrongPin, ack, cancel,
           fundsOK, problem, none};
ASSIGN
  init(state) := welcome;
  next(state) := case
    state = welcome & action = cardIn       : enterPin;
    state = enterPin & action = correctPin : askAmount ;
    state = enterPin & action = wrongPin   : tryAgain;
    state = tryAgain & action = ack        : enterPin;
    state = askAmount & action = fundsOK   : thanksGoodbye;
    state = askAmount & action = problem   : sorry;
    state = enterPin & action = cancel     : thanksGoodbye;
    TRUE                                   : state;
  esac;
LTLSPEC F( G state = thanksGoodbye
         | G state = sorry
        );
```

Running NuSMV

Batch

```
$ NuSMV atm.smv
```

Interactive

```
$ NuSMV -int atm.smv
NuSMV > go
NuSMV > check_ltlspec
NuSMV > quit
```

- `go` abbreviates the sequence of commands `read_model`, `flatten_hierarchy`, `encode_variables`, `build_model`.
- For command options, use `-h` or look in the NuSMV User Manual.

Expected Failure

```
NuSMV > check_ltlspec
-- specification  F ( G state = thanksGoodbye
                      |  G state = sorry)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = welcome
  input = cardIn
-> State: 1.2 <-
  state = enterPin
  input = correctPin
-- Loop starts here
-> State: 1.3 <-
  state = askAmount
  input = ack
-> State: 1.4 <-
```

Unexpected Failure

```
-- specification
    ( F ( G !(state = askAmount)) ->
      F ( G state = thanksGoodbye |  G state = sorry))
        is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  state = welcome
  input = cardIn
-- Loop starts here
-> State: 2.2 <-
  state = enterPin
  input = ack
-> State: 2.3 <-
```

Success

```
-- specification
     ( G (((state = welcome ->  F input = cardIn) &
           (state = enterPin ->
              F (state = enterPin &
             (input = correctPin | input = cancel)))) &
           (state = askAmount ->  F (input = fundsOK
                                 | input = problem))) ->
      F ( G state = thanksGoodbye |  G state = sorry))
    is true
```

Modules

```
MODULE counter
VAR digit : 0..9;
ASSIGN
  init(digit) := 0;
  next(digit) := (digit + 1) mod 10;

MODULE main
VAR c0 : counter;
    c1 : counter;
    sum : 0..99;
ASSIGN
    sum := c0.digit + 10 * c1.digit;
```

- Modules are instantiated in other modules. The instantiation is performed inside the VAR declaration of the parent module.
- In each SMV specification there must be a module main. It is the top-most module.
- All the variables declared in a module instance are visible in the module in which it has been instantiated via the dot notation (e.g., `c0.digit`, `c1.digit`).

Modules

```
MODULE counter
VAR digit : 0..9;
ASSIGN
  init(digit) := 0;
  next(digit) := (digit + 1) mod 10;

MODULE main
VAR c0 : counter;
    c1 : counter;
    sum : 0..99;
ASSIGN
    sum := c0.digit + 10 * c1.digit;

LTLSPEC
  F sum = 13;
```

- Is this specification satisfied by this model?

```
-- specification  F sum = 13  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  c0.digit = 0
  c1.digit = 0
  sum = 0
-> State: 1.2 <-
  c0.digit = 1
  c1.digit = 1
  sum = 11
-> State: 1.3 <-
  c0.digit = 2
  c1.digit = 2
  sum = 22
...
```

Modules with parameters

```
MODULE counter(inc)
VAR digit : 0..9;
ASSIGN
  init(digit) := 0;
  next(digit) := inc ? (digit + 1) mod 10
                     : digit;
DEFINE top := digit = 9;

MODULE main
VAR c0 : counter(TRUE);
    c1 : counter(c0.top);
    sum : 0..99;
ASSIGN
  sum := c0.digit + 10 * c1.digit;
```

- Formal parameters (inc) are substituted with the actual parameters (`TRUE`, `c0.top`) when the module is instantiated.
- Actual parameters can be any legal expression.
- Actual parameters are passed by reference.

```
-- specification  F sum = 13  is true
```