1 Introduction

Reliability: continuity of service

Availability: readiness for usage

Security: e.g. preservation of confidentiality

Safety: avoid catastrophic consequences

Fault prevention: system does not contain faults

Fault tolerance: system can recover from a system failure

Fail safety: in case of failure, no fatal consequences can occur

Preventing design bugs in hardware is important, because:

- costs are higher
- bug fixes are usually not possible
- quality expectations are higher
- time a new product gets on the market severely impacts revenue

Validation: Are we building the right thing?

Verification: Are we building the thing right?

Problem with testing: Can uncover faults, but doesn't guarantee their absence.

Formal verification can guarantee fault freeness.

2 Model Checking

Model: formal abstraction of the system of interest

Constraints: system requirements specified with precise logical statements A model checker automaticall checks the model against the constraints. If something violates the constraints, an execution path is returned.

Examples of system properties

- functional correctness: system behaving as expected?
- reachability: can we end up in a deadlock?
- safety: something "bad" never happens
- liveness: something "good" will eventually happen
- fairness: can, under certain conditions, an event occur repeatedly?
- real-time: is the system acting in time?

Must ensure that model is **as simle as possible**: prevent state explosion.

Livelock: a procass is always able to request resource, but always refused.

Unconditional fairness: Every process executes infinitely often

Strong fairness: Every process that can run, will run

Weak fairness: Every process that can continuously run from a certain point onwards, will execute infinitely often.

3 PROMELA: Process Meta Language

Spec language used in **Spin**. Used to check **concurrent systems**:

- interleaved processes (stuff behaving independently)
- shared global variables
- synchronous/asynchronous channels

Can label certain parts of the model that are of interest and then assert stuff only about those states.

Can check for desirable end states.

Checking liveness in Spin: annotate program with `progress[a-zA-Z0-9]*:` labels, and it will be checked if we pass through the labels infinitely often.

Trace constraint: can restrain the internal interleaving to only perform certain kinds of transitions (i.e., some channels only send/receive one kind of message, always alternating, etc.)
A trace **synchronises** with the spec

Never claim: define an undesired system behaviour and check if it ever holds, and reports if it does.
It *succeeds* if it terminates of passes through its accept labels infintely often. If no transition is possible in the never claim, the claim automaton will stop search in that branch and backtrack. A never claim is **interleaved** with the model spec. (It always executes before the next transition)

4 LTL: Linear Temporal Logic

All valid LTL formulae may be derived from the grammar

$$\phi ::= \text{true} \mid \text{false} \mid p \mid \neg \mid \wedge \mid \vee \mid (\phi) \mid \diamond \phi \mid \Box \phi \mid \phi \; \mathcal{U} \; \phi$$

where p is an **atomic proposition** (basically just a Boolean variable). They are **simple** known facts about the model at a particular state.

Examples

- Invariant: $\Box p$
- Reply: $p \Rightarrow \diamond q$
- Guaranteed Reply: $\Box(p \Rightarrow \mathcal{N} \; (\diamond q))$
- Infinitely often: $\Box \diamond p$
- Eventually always: $\diamond \Box p$
- Exclusion: $\Box(p \Rightarrow \neg q)$

Equivalences

- $\neg \Box p \equiv \diamond \neg p$
- $\neg \diamond p \equiv \Box \diamond \neg p$
- $\Box p \wedge q \equiv \Box p \wedge \Box q$
- $\diamond p \vee q \equiv \diamond p \vee \diamond q$
- $\diamond p \equiv \text{true} \; \mathcal{U} \; p$

Transition System a TS is defined as $TS = (S, Act, T, I, AP, L)$

- S: set of all possible states (nodes of the multigraph)
- Act: set of all possible transitions
- $T \subseteq S \times Act \times S$: transition table (edges of the multigraph)

- $I \subseteq S$: initial states
- AP: set of atomic propositions
- $L : S \to 2^{AP}$: labelling function (state labels)

A TS if **finite** iff S, Act and AP are finite. A state satisfies a formula if its label set satisfies it, i.e.

$$s \vDash \phi \equiv L(s) \vDash \phi$$

Definitions

- **direct α-successors of a state** s: $\text{Post}(s, \alpha) = \{s' \in S \mid (s, \alpha, s') \in T\}$ for $s \in S$ and $\alpha \in Act$. (States that are pointed to by an α arrow from s.)
- **α-predecessors of** s: $\text{Pre}(s, \alpha) = \{s' \in S \mid (s', \alpha, s) \in T\}$ for $s \in S$ and $\alpha \in Act$. (States that point to s with an α arrow from s.)
- **terminal state**: $s \in S$ is terminal iff $\text{Post}(s) = \emptyset$.
- **finite execution fragment (FEF)**: for a TS, a FEF is

  $$\rho = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$$

  where $s_i \in S$ and $\alpha_j \in Act$.
- **maximal execution fragment (MEF)**: FEF ending in a terminal state or an infinite EF.
- **initial execution fragment (IEF)**: EF whose first state is an initial state.
- **execution of a TS**: IEF + MEF
- **reachable state**: a state $s \in S$ is reachable iff

  $$\exists \rho. \quad \rho = s_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} s_n$$

  where $s_0 \in I$ and $s_n = s$ (you can transition into it in a finite amount of time).

5 Data-Dependent Systems

Want to deal with transitions dependent on some "global state". In a TS: use nondeterminism (this doesn't scale well), or **conditional transitions**.

Conditional transition: $g : \alpha$ where α is an action and g is a boolean condition (guard).

Definitions (bonkers)

- **set of typed variables**: `Var`
- **domain of a variable**: the type of $x \in$ `Var` is (inexplicably) called its domain, denoted $\text{dom}(x)$.
- **evaluations**: The set of **evaluations** over `Var` is the set that contains how we allow variables to evaluate and is denoted Eval(`Var`). e.g. we might say that x can evaluate to $1, 42$ and y can evaluate to "banana", "apple", respectively. Then we can define $\eta_1(x) = 1$, $\eta_1(y) =$ "banana"' $\eta_2(x) = 42$, $\eta_2(y) =$ "apple", and hence Eval(`Var`) $= \{\eta_1, \eta_2\}$.
- **condition set**: the **condition set** over `Var` is a set of boolean conditions involving the elements of `Var`.
- **Effect function**: indicates how variables evolve if we perform some action. Namely, Effect : $Act \times \text{Eval}(Var) \to \text{Eval}($`Var`$)$. e.g. if α represents incrementing x, and $\eta_i(x) = i$, then

  $$\text{Effect}(\alpha, \eta_i) = \eta_{i+1}.$$

  Since this is a function, we can evaluate it:

  $$\text{Effect}(\alpha, \eta_i)(x) = \eta_{i+1}(x) = i + 1.$$

- **Program Graph**: a PG is $PG = (Loc, Act, \text{Effect}, C_\dagger, Loc_0, g_0)$, where
  - Loc: set of locations, because the word state is not good anymore. (the justification being that the some location can be in different "states")
  - Act: Act is still good though, set of actions
  - Effect: see above
  - $C_\dagger$: $C_\dagger \subseteq Loc \times \text{Cond}($`Var`$) \times Act \times Loc$, the conditional transition table. A generic element looks like

    $$l_1 \xrightarrow{g:\alpha} l_2$$

    i.e. if g holds when we perform α in l_1, we transition into l_2.
  - Loc_0: $Loc_0 \in Loc$ set of initial locations
  - g_0: $g_0 \in \text{Cond}($`Var`$)$ is the initial condition (basically ensuring that our variables exist and have some initial value).

A PG can be converted into a TS, by "unfolding" it, i.e. adding a state for every location + evaluation combo, and a transition between them if we the guard of the transition connecting the two locations is true.

6 Parallelism

TSs are good for **sequential** systems. How do we model **parallel** systems? Depends on how *subsystem*s communticate: no communication, synchronous comm., asynchronous comm.

$$TS = TS_1 \mid\mid TS_2 \mid\mid \ldots \mid\mid TS_n$$

where $||$ is the **parallel composition operator**, meaning that the TS_i execute at the same time.

Interleaving is choosing the order of execution for the TS_i. Two TSs interleaved is the Cartesian product of their states and union of their action sets and atomic props, and is denoted $TS_1 ||| TS_2$, where $|||$ is called the **interleaving operator**. Note: interleaving doesn't reflect the expected behaviour of shared variables.
For programs with shared variables, we interleave PGs, and then unfold them.

Concurrency is nondeterministic interleaving

Independent actions are actions that can be interleaved, and their effect doesn't change

Local variables of PG_1 and PG_2 are $\texttt{Var}_1 \setminus \texttt{Var}_2$ and vice versa.

Global Variables are $\texttt{Var}_1 \cap \texttt{Var}_2$.

Note: Look at Lecture 8 - 10 for the graphs.