Testing from Formal Specifications, *a unifying framework*

Marie-Claude Gaudel
Emeritus Professor
LRI, Univ Paris-Sud & CNRS
Software Testing can be formal too

A pioneering paper:

• « We know less about the theory of testing, which we do often, than about the theory of program proving, which we do seldom »

Goodenough J. B., Gerhart S.,
IEEE Transactions on Software Engineering, 1975
There have been some progresses...

Outline of the course

• *Introduction Part*
  – Formal specifications
  – Testing

• Putting them together

• *Case splitting methods*
  – DNF, unfolding,…

• *Illustrations*
  – Axioms, FSM, CSP
1 - Formal Specifications?

- As for any specification framework, there is a notation, for instance:
  - Formulas
    - Pre/Post-conditions, 1st order logic, JML, SPEC# …
    - Algebraic Spec (CASL), Z, VDM, B,
  - Processes definitions
    - CSP, CCS, Lotos, Circus …
  - Annotated diagrams
    - Automata, Finite State Machines (FSM), Petri Nets…
- But there is more than a syntax…
What makes a specification method formal?

• There is a formal semantics
  – Algebras, Predicate transformers, Sets, Labelled Transition Systems (LTS), Traces and Failures…

• There is a formal system (proofs) or a verification method (model-checking), or both.

• Thus
  – Formal specifications can be analysed to guide the identification of appropriate test cases.
  – They may contribute to the definition of oracles.
Example 1: Pre/Post-conditions (à la VDM)

MAX (a:Z, b:Z)

result max:Z

pre true

post (max=a ∨ max=b) ∧ max≥a ∧ max≥b
Example 1bis: axioms of a data type (à la CASL)

\[\text{spec} \ \text{Container} = \text{NAT}, \text{BOOL}\]

\[\text{then}\]

\[\text{generated type} \ \text{Container ::= [ ] | _::_ (Nat ; Container)}\]

\[\text{op isin : Nat } \times \text{ Container } \rightarrow \text{Bool}\]

\[\text{op remove: Nat } \times \text{ Container } \rightarrow \text{Container}\]

\[\forall x, y: \text{Nat; } c: \text{Container}\]

\[\bullet \ \text{isin}(x, [ ]) = \text{false}\]

\[\bullet \ \text{eq}(x, y) = \text{true} \Rightarrow \text{isin}(x, y::c) = \text{true}\]

\[\bullet \ \text{eq}(x, y) = \text{false} \Rightarrow \text{isin}(x, y::c) = \text{isin}(x,c)\]

\[\bullet \ \text{remove}(x, [ ]) = [ ]\]

\[\bullet \ \text{eq}(x, y) = \text{true} \Rightarrow \text{remove}(x, y::c) = c\]

\[\bullet \ \text{eq}(x, y) = \text{false} \Rightarrow \text{remove}(x, y::c) = y::\text{remove}(x,c)\]

\[\text{end}\]
This FSM removes from the input text all that is not a comment
A comment is a string between /* and */
Examples:
This is not a comment /* all that / *is ** a comment */ this is no more a comment.
NB: $\phi$ is any character but * and /
Example 3: CSP processes

\[
\begin{align*}
\text{Counter}_2 &= \text{add} \rightarrow C_1 \\
C_1 &= \text{add} \rightarrow C_2 [ \quad ] \text{sub} \rightarrow \text{Counter}_2 \\
C_2 &= \text{sub} \rightarrow C_1
\end{align*}
\]

\[
\begin{align*}
\text{Replicator} &= c? x : \text{Int} \rightarrow d! x \rightarrow \text{Replicator} \\
\text{FreshInt}(n : \text{Int}) &= c! n \rightarrow \text{FreshInt}(n + 1) \\
(FreshInt(0) \mid [c] \mid \text{Replicator}) \setminus c &\quad \text{parallel composition with hidden synchronisation on } c
\end{align*}
\]
Example 3bis: a Circus process

\[
\text{RANGE} \equiv \{0..59\}
\]

channel \text{tick, time}

channel \text{out : RANGE \times RANGE}

process \text{Chrono} \equiv \begin{array}{l}
\text{state AState} \equiv \{\text{sec, min : RANGE}\} \\
\text{AINit} \equiv \{\text{AState'} | \text{sec'} = \text{min'} \land \text{min'} = 0\} \\
\text{IncSec} \equiv \{\Delta\text{AState} | \text{sec'} = (\text{sec} + 1) \mod 60 \land \text{min'} = \text{min}\} \\
\text{IncMin} \equiv \{\Delta\text{AState} | \text{min'} = (\text{min} + 1) \mod 60 \land \text{sec'} = \text{sec}\}
\end{array}

\begin{align*}
\text{Run} & \equiv \text{tick} \rightarrow \text{IncSec}; ((\text{sec} = 0) \land \text{IncMin}) \\
& \quad \text{□} \\
& \quad (\text{sec} \neq 0) \land \text{Skip})
\end{align*}

\begin{align*}
\text{time} & \rightarrow \text{out}!(\text{min, sec}) \rightarrow \text{Skip} \\
\bullet & (\text{AINit}; (\mu X \bullet (\text{Run}; X)))
\end{align*}

end

process \text{Clock} \equiv \begin{array}{l}
\bullet \mu X \bullet \text{tick} \rightarrow X
\end{array}

process \text{TChrono} \equiv (\text{Chrono \{\{\text{tick}\}\}} \text{Clock} \setminus \{\{\text{tick}\}\})
2 - Testing

• One tests SYSTEMS
• A system is a dynamic entity, *embedded in the physical world*
• It is *observable* via some limited interface/procedure
• It is not always *controllable*
• It is quite different from a piece of text (formula, program) or a diagram
A philosophical interlude

“A map is not the territory”* Korzybski

*A variant: “don’t eat the menu…” 😊

A program text, or a specification text, or a model, is not the system
Black-Box Testing

• **Black-Box Testing:**
  – the internal organisation of the SUT (System Under Test) is not known

• **However,**
  – Implicitely or explicitely, one considers a class of “testable implementations” => notion of *Testability Hypotheses* on the SUT
Testability?

- If the SUT can be *any demonic system*, there is no sensible way of testing it 😞
- Fortunately, *some basic assumptions are feasible* (example: correct implementation of booleans and bounded integers, determinism, …)
- Some others can be *verified in another way*: static checks on the program, preliminary tests, a priori knowledge of the environment…
Wanted: a satisfaction/conformance relation

Given some “testable” SUT, what does it mean that it satisfies SP?

What is the correctness reference? Is there an “exhaustive” (or “complete”) set of tests?

SP is some sort of model or formula; SUT is some sort of system; how to define “SUT sat SP” or “SUT conf SP” in such an heterogeneous context?
A generic testability hypothesis

• “The SUT corresponds to some unknown formal specification in the same formalism as specification SP”
  – If SP is a FSM, SUT behaves like some FSM
  – If SP is a formula, the symbols of the formula can be interpreted by SUT
  – If SP is a process, SUT can be observed as a process, with traces and deadlocks

• Notation: \([SUT]\)
For instance, the *satisfaction/conformance* relation is
- equivalence for FSM,
- logical satisfaction for formulas,
- refinement for processes,
- *ioco* for LTS…
Tests and Test drivers

consequences, counter-examples

sat/conf/refines

SP

SUT
Illustration: testing against traces refinement in CSP

\[
\begin{align*}
\text{Counter}_2 &= \text{add} \rightarrow C_1 \\
C_1 &= \text{add} \rightarrow C_2 \left[ \text{sub} \rightarrow \text{Counter}_2 \right] \\
C_2 &= \text{sub} \rightarrow C_1
\end{align*}
\]

Traces of \(\text{Counter}_2\):

\[
\langle \rangle, \langle \text{add} \rangle, \langle \text{add,add} \rangle, \langle \text{add,sub} \rangle, \langle \text{add,add,sub} \rangle, \ldots
\]
Illustration: testing against traces refinement in CSP

\[ \text{Counter}_2 = \text{add} \rightarrow C_1 \]
\[ C_1 = \text{add} \rightarrow C_2 \text{[ sub }] \rightarrow \text{Counter}_2 \]
\[ C_2 = \text{sub} \rightarrow C_1 \]

Forbidden traces
\[ <\text{sub}> \]
\[ <\text{add,add}> \]
\[ <\text{add,sub}> \]
\[ ... \]

Traces of \text{Counter}_2
\[ <> \]
\[ <\text{add}> \]
\[ <\text{add,add}> \]
\[ <\text{add,sub}> \]
\[ <\text{add,add,sub}> \]
\[ ... \]

\[ \text{test1} = \text{pass} \rightarrow \text{sub} \rightarrow \text{fail} \rightarrow \text{STOP} \]
\[ \text{test2} = \text{inc} \rightarrow \text{add} \rightarrow \text{inc} \rightarrow \text{add} \rightarrow \text{pass} \rightarrow \text{add} \rightarrow \text{fail} \rightarrow \text{STOP} \]
\[ \text{test3} = \text{inc} \rightarrow \text{add} \rightarrow \text{inc} \rightarrow \text{sub} \rightarrow \text{pass} \rightarrow \text{sub} \rightarrow \text{fail} \rightarrow \text{STOP} \]
Illustration: testing against traces refinement in CSP

\[ \text{Counter}_2 = \text{add} \rightarrow C_1 \]
\[ C_1 = \text{add} \rightarrow C_2 [ \text{sub} \rightarrow \text{Counter}_2 ] \]
\[ C_2 = \text{sub} \rightarrow C_1 \]

Forbidden traces
- \(<\text{sub}>\)
- \(<\text{add,add,add}>\)
- \(<\text{add,sub,sub}>\)
- ...

Traces of \text{Counter}_2
- \(<>\)
- \(<\text{add}>\)
- \(<\text{add,add}>\)
- \(<\text{add,sub}>\)
- \(<\text{add,add,sub}>\)
- ...

Test submissions
- \(SUT | [\text{add,sub}] | \text{test1} \setminus [\text{add,sub}]\)
- \(SUT | [\text{add,sub}] | \text{test2} \setminus [\text{add,sub}]\)
- \(SUT | [\text{add,sub}] | \text{test3} \setminus [\text{add,sub}]\)

Oracle: the last observed event is not \textit{fail}
Exhaustive test set for traces refinement of CSP

Let us consider the Test Set:

\[ \text{Exhaust}_T (SP) = \{ T_T (s , a) \mid s \in \text{traces} (SP) \land \neg a \in \text{initials} (SP /s) \} \]

where

\[ T_T (s , a) = \text{inc} \rightarrow a_1 \rightarrow \text{inc} \rightarrow a_2 \rightarrow \text{inc} \ldots a_n \rightarrow \text{pass} \rightarrow a \rightarrow \text{fail} \rightarrow \text{STOP} \]

for \( s = <a_1,a_2, ...,a_n> \).

For any test \( T \), its execution against \( SUT \) is specified as:

\[ \text{Execution}_{SP,SUT} (T) = (SUT \mid [aSP ]\mid T)\backslash aSP \]

Theorem (Cavalcanti Gaudel 2007) :

\( SUT \) is a traces refinement of \( SP \) iff

\[ \forall T_T (s , a) \in \text{Exhaust}_T (SP), \forall t \in \text{traces} (\text{Execution}_{SP,SUT} (T_T (s , a))), \neg \text{last} (t) = \text{fail} \]
The corresponding testability hypotheses

• *SUT* behaves like a CSP process
  – With the same alphabet of actions as *SP*
  – The *actions and events are atomic*

• If *SUT* is non-determinist, it satisfies the classical *complete testing assumption*
Exhaustivity is not practicable

exhaust(SP) ?

You are not serious!

Let us select some adequate finite subset

It has been my problem for years…
Selection

• How to select finite subsets of $\text{Exhaust}_{SP}$?

• Test Set Selection is based on the specification (of course, it’s Black Box Testing!)

• Among the solutions:
  – Uniformity hypotheses
  – Regularity hypotheses
  – Others …
An example from CSP

Replicator = c? x : Int → d! x → Replicator

FreshInt(n : Int) = c! n → FreshInt(n + 1)

(FreshInt(0) | [c] | Replicator) \ c  parallel composition

with hidden synchronisation on c

Forbidden symbolic traces
< d.v >  ∀ v ∈ Int
< c.v, d.w >  ∀ v, w ∈ Int, v ≠ w
< c.v, c.w >  ∀ v, w ∈ Int
< c.v, d.v, d.w >  ∀ v, w ∈ Int
< c.v, d.v, c.w, d.u >  ∀ v, w, u ∈ Int, w ≠ u
...

Traces of Replicator
<>
<c.0>
<c.1>...
<c.0,d.0>
<c.1,d.1>...
<c.0,d.0,c.7>...
An example from CSP

\[ \text{Replicator} = c? x : \text{Int} \rightarrow d! x \rightarrow \text{Replicator} \]

\[ \text{FreshInt}(n : \text{Int}) = c! n \rightarrow \text{FreshInt}(n+1) \]

\( (\text{FreshInt}(0) \mid [c] \mid \text{Replicator}) \setminus c \) parallel composition with hidden synchronisation on \( c \)

Forbidden symbolic traces

\[ <d.v> \quad \forall \ v \in \text{Int} \]

\[ <c.v, d.w> \quad \forall \ v, w \in \text{Int}, \ v \neq w \]

\[ <c.v, c.w> \quad \forall \ v, w \in \text{Int} \]

\[ <c.v, d.v, d.w> \quad \forall \ v, w \in \text{Int} \]

\[ <c.v, d.v, c.w, d.u> \forall \ v, w, u \in \text{Int}, \ w \neq u \]

No condition on \( v \): an arbitrary value will do => Uniformity Hypothesis

There is one condition on \( w \): \( v \neq w \). Any value satisfying it will do => Uniformity Hypothesis, etc
An example from CSP

\[ \text{Replicator} = c?x : \text{Int} \rightarrow d!x \rightarrow \text{Replicator} \]

\[ \text{FreshInt}(n : \text{Int}) = c!n \rightarrow \text{FreshInt}(n+1) \]

\((\text{FreshInt}(0) \mid [c] \mid \text{Replicator}) \setminus c\) parallel composition with hidden synchronisation on \(c\)

Forbidden symbolic traces
\[<d.v> \ \forall \ v \in \text{Int}\]
\[<c.v, d.w> \ \forall \ v,w \in \text{Int}, v \neq w\]
\[<c.v, c.w> \ \forall \ v,w \in \text{Int}\]
\[<c.v, d.v, d.w> \ \forall \ v,w \in \text{Int}\]
\[<c.v, d.v, c.w, d.u> \ \forall \ v,w,u \in \text{Int}, w \neq u\]

No condition on \(v\): an arbitrary value will do
\[\Rightarrow \text{Uniformity Hypothesis} \Rightarrow \text{test1}\]
There is one condition on \(w\): \(v \neq w\). Any value satisfying it will do
\[\Rightarrow \text{Uniformity Hypothesis} \Rightarrow \text{test2}, \text{ etc}\]

\[\text{test1} = \text{pass} \rightarrow d.127 \rightarrow \text{fail} \rightarrow \text{STOP}\]
\[\text{test2} = \text{inc} \rightarrow c.0 \rightarrow \text{pass} \rightarrow d.17 \rightarrow \text{fail} \rightarrow \text{STOP}\]
\[\text{test3} = \text{inc} \rightarrow c.4 \rightarrow \text{pass} \rightarrow c.1024 \rightarrow \text{fail} \rightarrow \text{STOP}\]
\[\text{test4} = \text{inc} \rightarrow c.78 \rightarrow \text{inc} \rightarrow d.78 \rightarrow \text{pass} \rightarrow d.46 \rightarrow \text{fail} \rightarrow \text{STOP}\]
\[\text{test5} = \ldots\]
But this test set is still infinite!!
And by the way, are you sure that test5 would be useful?

Just make use of regularity…but it is sometimes risky.

What a crazy academic!
An example of regularity hypothesis

\[ \text{Replicator} = c? x : \text{Int} \rightarrow d!x \rightarrow \text{Replicator} \]

\[ \text{FreshInt}(n : \text{Int}) = c!n \rightarrow \text{FreshInt}(n + 1) \]

\((\text{FreshInt}(0) \mid [c] \mid \text{Replicator}) \setminus c\) parallel composition with hidden synchronisation on \(c\)

Forbidden symbolic traces

\(<d.v> \forall v \in \text{Int} >\)

\(<c.v, d.w> \forall v, w \in \text{Int}, v \neq w >\)

\(<c.v, c.w> \forall v, w \in \text{Int} >\)

\(<c.v, d.v, d.w> \forall v, w \in \text{Int} >\)

\(<c.v, d.v, c.w, d.u> \forall v, w, u \in \text{Int}, w \neq u >\)

There is no dependency between the recursive calls of \(\text{Replicator}\).

There is no shared state.

⇒ If the SUT is determinist, one execution is sufficient => Regularity Hypothesis => Finite Test Set

\[
\begin{align*}
test1 &= \text{pass} \rightarrow d.127 \rightarrow \text{fail} \rightarrow \text{STOP} \\
test2 &= \text{inc} \rightarrow c.0 \rightarrow \text{pass} \rightarrow d.17 \rightarrow \text{fail} \rightarrow \text{STOP} \\
test3 &= \text{inc} \rightarrow c.4 \rightarrow \text{pass} \rightarrow c.1024 \rightarrow \text{fail} \rightarrow \text{STOP} \\
test4 &= \text{inc} \rightarrow c.78 \rightarrow \text{inc} \rightarrow d.78 \rightarrow \text{pass} \rightarrow d.46 \rightarrow \text{fail} \rightarrow \text{STOP}
\end{align*}
\]
Selection Hypotheses

• Addition to Testability Hypotheses: Selection Hypotheses on the SUT

• Uniformity Hypothesis
  – \( \Phi(X) \) is a property, \( SUT \) is a system, \( D \) is a sub-domain of the domain of \( X \)
  – \( (\forall t_0 \in D) \left( [SUT] \text{ sat } \Phi(t_0) \Rightarrow (\forall t \in D) ([SUT] \models \Phi(t)) \right) \)
  – Determination of sub-domains? guided by the specification, see later...

• Regularity Hypothesis
  – \( ( (\forall t \in \text{Dom}(X), |t| \leq k \Rightarrow [SUT] \text{ sat } \Phi(t))) \Rightarrow \\
    (\forall t \in \text{Dom}(X) ([SUT] \text{ sat } \Phi(t)) \)
  – Determination of \(|t|\)? cf. specification
Selection of finite test sets

- “Selection Hypotheses” $H$ on $SUT$, and construction of practicable test sets $T$ such that:

  $H$ holds for $SUT => (SUT$ passes $T <=> [SUT]$ sat $SP)$

- $<H, T>$ is a valid and unbiased Test Context

- or: $T$ is complete w.r.t. $H$

<SUT testable, exhaust(SP)>

<Weak Hyp, Big Test Set>

<Strong Hyp, Small TS>

<SUT correct, Ø>
SOME BASIC TECHNIQUES FOR CASE SPLITTING
“Invention” of selection hypotheses

Several possibilities:

• Guided by the conditions that appear in the specification: case analysis, case splitting
• Or guided by some knowledge of the operational environment
• Or guided by some fault model
• Or guided by the syntax (coverage criteria)
Case splitting

Two main techniques:

• Reduction of formulas into **Disjunctive Normal Form** (DNF) [*Dick & Faivre 1993*]
• **Unfolding** of recursive definitions [*Burstall & Darlington 1977*]

Implementations:

• Conditional rewriting, Narrowing
• Symbolic evaluation
DNF?

- It is a disjunction (sequence of ORs) consisting of one or more disjuncts, each of which is a conjunction (AND) of one or more literals (i.e., statement letters and negations of statement letters; Mendelson 1997, p. 30)

- $\land$, $\lor$, and $\neg$ are the only logical operators, $\neg$ is the most internal, then $\land$, then $\lor$

- Intuitively, this gives a list of disjoint test cases.
More on DNF

• A first example of DNF decomposition:

\[(p \lor q) \rightarrow \neg r \iff (p \land q \land \neg r) \lor (p \land \neg q \land \neg r) \lor (\neg p \land q \land \neg r) \lor (\neg p \land \neg q \land r) \lor (\neg p \land \neg q \land \neg r)\]

• Basic rules:

  – \((p \lor q)\) is decomposed into 3 disjoint cases: \(p \land q\), \(p \land \neg q\), \(\neg p \land q\)

  – \((A \rightarrow B)\) is decomposed into \(\neg A\) and \(A \land B\)

  – \(\neg \neg \) are eliminated

• Not very difficult, but… exponential explosion
Example of the reduction of pre/post-conditions

\[
\text{MAX (a:}\mathbb{Z}, \text{ b:}\mathbb{Z}) \\
\text{result max:}\mathbb{Z} \\
\text{pre true} \\
\text{post} (\text{max=a }\lor \text{ max=b}) \land \text{ max}\geq a \land \text{ max}\geq b
\]

(conjunction of pre-condition, post-condition and state Invariant, if any) =>

\[
\text{true }\land (\text{max=a }\lor \text{ max=b}) \land \text{ max}\geq a \land \text{ max}\geq b
\]
Example of the reduction of pre/post-conditions

MAX (a: Z, b: Z)
result max: Z
pre true
post (max = a ∨ max = b) ∧ max ≥ a ∧ max ≥ b

true ∧ (max = a ∨ max = b) ∧ max ≥ a ∧ max ≥ b
(conjunction of pre-condition, post-condition and state Invariant, if any) =>

(max = a ∨ max = b) ∧ max ≥ a ∧ max ≥ b
(simplification of “true ∧ …”) =>
Example of the reduction of pre/post-conditions

MAX (a: Z, b: Z)
result max: Z
pre true
post (max = a ∨ max = b) ∧ max ≥ a ∧ max ≥ b

true ∧ (max = a ∨ max = b) ∧ max ≥ a ∧ max ≥ b (simplification of “true ∧ …”) =>

(max = a ∨ max = b) ∧ max ≥ a ∧ max ≥ b (distribution of ∨ ) =>

(max = a ∧ max ≥ a ∧ max ≥ b) ∨ (max = b ∧ max ≥ a ∧ max ≥ b)
Example of the reduction of pre/post-conditions

\[
\text{MAX (a:Z, b:Z)}\hspace{1cm} \text{(conjunction of pre-condition, post-condition and state Invariant, if any) =>}
\]

\[
\text{true } \land (\max=a \lor \max=b) \land \max \geq a \land \max \geq b \hspace{1cm} \text{(simplification of “true } \land \ldots”) \Rightarrow
\]

\[
(\max=a \lor \max=b) \land \max \geq a \land \max \geq b \hspace{1cm} \text{(distribution of } \lor ) \Rightarrow
\]

\[
(\max=a \land \max \geq a \land \max \geq b) \lor (\max=b \land \max \geq a \land \max \geq b) \hspace{1cm} \text{(decomposition of } \lor ) \Rightarrow
\]

\[
(\max=a \land \max=b \land \max \geq a \land \max \geq b) \lor \\
(\max=a \land \max \neq b \land \max \geq a \land \max \geq b) \lor \\
(\max \neq a \land \max=b \land \max \geq a \land \max \geq b)
\]
Example of the reduction of pre/post-conditions

\[
\begin{align*}
\text{MAX} \ (a: \mathbb{Z}, b: \mathbb{Z}) \\
\text{result} \ max: \mathbb{Z} \\
\text{pre} \ true \\
\text{post} \ (\max = a \lor \max = b) \land \max \geq a \land \max \geq b
\end{align*}
\]

(conjunction of pre-condition, post-condition and state Invariant, if any) =>

\[
\begin{align*}
\text{true} \land (\max = a \lor \max = b) \land \max \geq a \land \max \geq b
\end{align*}
\]

(simplification of “true \land…”) =>

\[
\begin{align*}
(\max = a \lor \max = b) \land \max \geq a \land \max \geq b
\end{align*}
\]

(distribution of \lor) =>

\[
\begin{align*}
(\max = a \land \max \geq a \land \max \geq b) \lor (\max = b \land \max \geq a \land \max \geq b)
\end{align*}
\]

(decomposition of \lor) =>

\[
\begin{align*}
(\max = a \land \max = b \land \max \geq a \land \max \geq b) \lor \\
(\max = a \land \max \neq b \land \max \geq a \land \max \geq b) \lor \\
(\max \neq a \land \max = b \land \max \geq a \land \max \geq b)
\end{align*}
\]

(simplifications) =>

\[
\begin{align*}
(\max = a \land \max = b) \lor \\
(\max = a \land \max > b) \lor \\
(\max = b \land \max > a)
\end{align*}
\]

3 test cases: \{(a=\max = a=b), (a>b, \max = a), (b>a, \max = b)\}

Thus, \textit{3 uniformity sub-domains + oracles.}
Unfolding

- Unfolding is a classical technique for transforming (and understanding) recursive definitions
- It is just replacement of \( f(op(x)) \) by the definition(s) of \( f \), with adequate renaming of variables
  - \( \text{fact}(n) = \text{def}\ if\ n=0\ then\ 1\ else\ n*\text{fact}(n-1) \) becomes:
  - \( \text{fact}(n) = \text{def}\ if\ n=0\ then\ 1\ else\ if\ (n-1)=0\ then\ n*1\ else\ n*(n-1)*\text{fact}(n-2) \)
    - i.e. \( \text{fact}(n) = \text{def}\ if\ n=0\ then\ 1\ else\ if\ n=1\ then\ 1\ else\ n*(n-1)*\text{fact}(n-2) \)
  - etc
  - Going on, the definition of the \text{fact} function is replaced by its graph, i.e. its exhaustive test set 😊…