

Testing, Verification, and Validation

Friedrich v. Henke¹ Cinzia Bernardeschi² Paolo Masci²
Holger Pfeifer¹ Hélène Waeselynck³

¹Universität Ulm

²Università di Pisa

³LAAS-CNRS



Part I: Model Checking, Friedrich v. Henke

Part II: Theorem Proving, Friedrich v. Henke and Holger Pfeifer

Part III: Static Program Analysis, Cinzia Bernardeschi and Paolo Masci

Part IV: Introduction to Software Testing, Hélène Waeselynck

Part I

Model Checking

Friedrich v. Henke

Verification by Model Checking

Verification involves the checking or demonstration that an entity (a system, a program etc.) has certain properties.

Verification requires

- a description of the entity under consideration;
- a specification of the property to be verified;
- a methods of examining the entity with respect to the specified property.

For *model checking*:

- a description in form of a *state transition system*, typically an *abstraction* of the concrete system or program.
- the property is specified by a formula of a *temporal logic*,
- the method of examination is given by an algorithms for checking whether state transition system is a *model* of the specification formula in the sense of formal logic.

State Transition System

State transition systems serve as the “models” in model checking.

Formal characterization of a state transition systems:

$\mathcal{M} = (S, \rightarrow, L)$ with

- S finite set of states
- \rightarrow transition relation, i.e. $\rightarrow \subseteq S \times S$
such that for every $s \in S$ there is a $s' \in S$ with $s \rightarrow s'$ (*)
- L Labelling of states:
 $L : S \rightarrow \mathcal{P}(A)$, where A is a set of atomic labels

Intuitive meaning of labelling: every state is labelled by a set of atoms (a conjunction of atomic propositions) that are true in that state.

Condition (*) ensures that each state has (at least) one successor state (there are no “deadlocks”). This can always be achieved by adding a new state s_d , a transition $s_d \rightarrow s_d$ (a loop) and sufficiently many transitions $s_i \rightarrow s_d$.

Temporal Logics

Temporal logics permit us to express properties of sequences of *discrete* points of time or states; i.e. contrary to what the term may suggest they do *not* deal with continuous time.

The notion of truth is *dynamic*: in different states (i.e. at different points in time) propositions may have different truth values.

In the context of model checking, two notions of time and corresponding temporal logics are commonly used:

- *Branching time*): Different possible future states are considered.
 \rightsquigarrow *CTL – Computation Tree Logic*.
- *Linear time*: A linear time line, represented by a totally order sequence of points in time is considered.
 \rightsquigarrow *LTL – Linear-time Temporal Logic*

This course presents first CTL.

CTL: Syntax

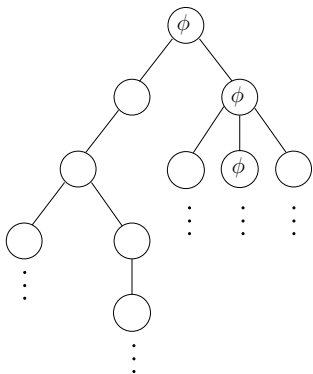
The structure of CTL formulas is defined by the following BNF:

$$\begin{aligned}\phi ::= & \perp \mid \top \mid p \mid \\ & (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid \\ & \mathbf{AX} \phi \mid \mathbf{EX} \phi \mid \mathbf{AG} \phi \mid \mathbf{EG} \phi \mid \mathbf{AF} \phi \mid \mathbf{EF} \phi \mid \\ & \mathbf{A}[\phi \mathbf{U} \phi] \mid \mathbf{E}[\phi \mathbf{U} \phi]\end{aligned}$$

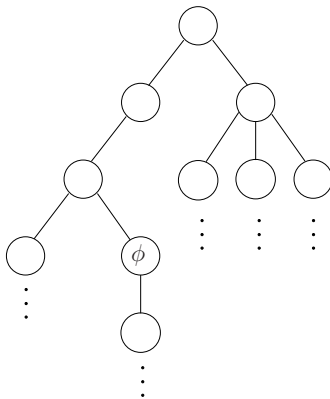
\top und \perp represent the constants *true* and *false*; p represents an atomic formula; the connectives are the usual ones from Propositional Logic.

The symbols **AX** etc. are *temporal operators*; each consists of two parts:

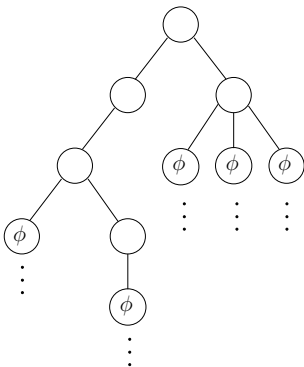
- **A** (“*Always*”) and **E** (“*Exists*”) are kinds of quantifier over paths.
- **X** (“*neXt*”), **F** (“*Future*”), **G** (“*Globally*”) and **U** (“*Until*”) refer to states along a path.



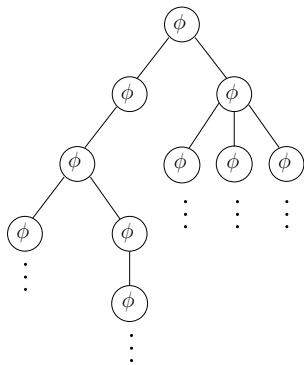
EG ϕ – "There exists a path such that ϕ holds at every state along that path"



EF ϕ – "There exists a path and a state along that path such that ϕ holds"



AF ϕ – "Along every path there exists a state such that ϕ holds at that state"



AG ϕ – "Along every path ϕ holds at every state"

Semantics of CTL

The semantics of CTL formulas is defined by reference to state transition systems as models.

The behaviour of a system is described by the possible execution paths, i.e. sequence of transitions.

A *path* π of \mathcal{M} is a sequence of states in S ,

$$s_1, s_2, \dots, s_i, s_{i+1}, \dots$$

such that s_{i+1} is a successor state of s_i for every $i \geq 1$, i.e. $s_i \rightarrow s_{i+1}$ for every i

The condition on the transition relation ensures that all paths are *infinite*.

A formula ϕ holding in a state s of model \mathcal{M} is expressed by:

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

Semantics of CTL (2)

The relation \models is defined inductively over the structure of CTL formulas.

- $\mathcal{M}, s \models \top$ and $\mathcal{M}, s \not\models \perp$ for all $s \in S$
- $\mathcal{M}, s \models p$ iff $p \in L(s)$
- $\mathcal{M}, s \models \neg\phi$ iff $\mathcal{M}, s \not\models \phi$
- $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \models \phi_1$ and $\mathcal{M}, s \models \phi_2$

Similarly for $\phi_1 \vee \phi_2$ and $\phi_1 \Rightarrow \phi_2$

- $\mathcal{M}, s \models \mathbf{AX} \phi$ iff $\mathcal{M}, s' \models \phi$ for every s' with $s \rightarrow s'$

AX: “in every successor state”

- $\mathcal{M}, s \models \mathbf{EX} \phi$ iff $\mathcal{M}, s' \models \phi$ for some s' with $s \rightarrow s'$

EX: “in some successor state”

Semantics of CTL (3)

- $\mathcal{M}, s \models \mathbf{AG} \phi$ iff $\mathcal{M}, s_i \models \phi$ holds for every path $\pi : s = s_1 \rightarrow s_2 \rightarrow \dots$ and every s_i on π

AG: “On every path starting at s and at every state on that path”

- $\mathcal{M}, s \models \mathbf{EG} \phi$ iff there exists a path $\pi : s = s_1 \rightarrow s_2 \rightarrow \dots$ such that $\mathcal{M}, s_i \models \phi$ for every state s_i on π .

EG: “There exists a path such that in every state on that path”

- $\mathcal{M}, s \models \mathbf{AF} \phi$ iff on every path $s = s_1 \rightarrow s_2 \rightarrow \dots$ there exists a s_i such that $\mathcal{M}, s_i \models \phi$

AF: “for every path starting at s there exists a future state on the path such that \dots holds”

Semantics of CTL (4)

- $\mathcal{M}, s \models \mathbf{EF} \phi$ iff there exists a path $\pi : s = s_1 \rightarrow s_2 \rightarrow \dots$ and a state s_i on π such that $\mathcal{M}, s_i \models \phi$

EF: “There exists a path starting at s and a future state on that path such that ...”

- $\mathcal{M}, s \models \mathbf{A} [\phi_1 \mathbf{U} \phi_2]$ iff every path $s = s_1 \rightarrow s_2 \rightarrow \dots$ satisfies $\phi_1 \mathbf{U} \phi_2$, i.e.

there exists an s_i on the path such that $\mathcal{M}, s_i \models \phi_2$ and $\mathcal{M}, s_j \models \phi_1$ for every s_j with $j < i$

AU: “For all paths starting at s ϕ_1 holds until ϕ_2 holds in a state.”

- $\mathcal{M}, s \models \mathbf{E} [\phi_1 \mathbf{U} \phi_2]$ iff there exists a path $s \rightarrow s_2 \rightarrow \dots$ which satisfies $\phi_1 \mathbf{U} \phi_2$ (as above).

EU: “There exists a path starting at s on which ϕ_1 holds (in every state) until a state is reached at which ϕ_2 holds.”

Equivalences of CTL Formulas

Two CTL formulas ϕ and ψ are semantically *equivalent* if they hold in the same states of a model:

$$\mathcal{M}, s \models \phi \text{ if and only if } \mathcal{M}, s \models \psi$$

Notation: $\phi \Leftrightarrow \psi$

Some important equivalences:

- The usual propositional equivalences also hold for CTL subformulas.
- “deMorgan-Regeln”: **A** und **E** bzw. **G** und **F** can be regarded as universal and existential quantors, respectively, over paths and the states along paths, respectively.

Hence the following relationships among operators hold:

$$\neg \mathbf{AF} \phi \Leftrightarrow \mathbf{EG} \neg \phi \qquad \neg \mathbf{EF} \phi \Leftrightarrow \mathbf{AG} \neg \phi \qquad \neg \mathbf{AX} \phi \Leftrightarrow \mathbf{EX} \neg \phi$$

Equivalences of CTL Formulas (2)

Relationships between **AF** , **AU**, **EF** and **EU**:

$$\mathbf{AF} \phi \Leftrightarrow \mathbf{A} [\top \mathbf{U} \phi] \qquad \mathbf{EF} \phi \Leftrightarrow \mathbf{E} [\top \mathbf{U} \phi]$$

Characterisation of temporal CTL operators by fixed points:

$$\mathbf{AG} \phi \Leftrightarrow \phi \wedge \mathbf{AX} \mathbf{AG} \phi$$

$$\mathbf{EG} \phi \Leftrightarrow \phi \wedge \mathbf{EX} \mathbf{EG} \phi$$

$$\mathbf{AF} \phi \Leftrightarrow \phi \vee \mathbf{AX} \mathbf{AF} \phi$$

$$\mathbf{EF} \phi \Leftrightarrow \phi \vee \mathbf{EX} \mathbf{EF} \phi$$

$$\mathbf{A} [\phi \mathbf{U} \psi] \Leftrightarrow \psi \vee (\phi \wedge \mathbf{AX} \mathbf{A} [\phi \mathbf{U} \psi])$$

$$\mathbf{E} [\phi \mathbf{U} \psi] \Leftrightarrow \psi \vee (\phi \wedge \mathbf{EX} \mathbf{E} [\phi \mathbf{U} \psi])$$

Reduced Sets of CTL Operators

The relationships among CTL operators indicate how operators may be replaced by combinations of others, thereby reducing the number of required operators.

$$\mathbf{AX} \rightsquigarrow \neg \mathbf{EX} \neg$$

$$\mathbf{AG} \phi \rightsquigarrow \neg \mathbf{EF} \neg \phi \rightsquigarrow \neg(\mathbf{E} [\top \mathbf{U} \neg \phi])$$

$$\mathbf{EG} \phi \rightsquigarrow \neg \mathbf{AF} \neg \phi \rightsquigarrow \neg(\mathbf{A} [\top \mathbf{U} \neg \phi])$$

This shows that the operators **AU**, **EU** and **EX** are sufficient.

Examples of other sufficient operator sets:

EG, EU, EX

AG, AU, AX

AF, EU, EX

Patterns of CTL Specifications

Certain forms of formulas occur frequently in specifications of practically relevant properties; they can be identified as specification patterns.

Examples include:

“For any state, when a device is *requested* it will also be *ready* eventually”:

AG (*requested* \Rightarrow **AF** *ready*)

“*ready* (e.g. for a process) is true infinitely often on every execution path”:

AG (**AF** *ready*)

“*terminated* (e.g. for a process) will be reached in any case”:

AF (**AG** *terminated*)

“It is always (i.e. from every state) possible to reach the start state”:

AG (**EF** *start*)

Example: Mutual Exclusion

Mutual Exclusion is an important principle for concurrent processes: it is to prevent several processes from simultaneously accessing a shared critical resource.

A common implementation of mutual exclusion is by having a *critical region* in each of the respective programs; a protocol determines under which conditions a program may enter its critical region.

We assume that execution of processes is interleaved: each process performs state transitions separately and independently, but not simultaneously with others.

This and subsequent examples, as well as most figures, are taken from M. Huth, M. Ryan: Logic in Computer Science.

Mutual Exclusion: Properties

Required properties for mutual exclusion:

- ① *Safety*: At any point of time, at most one process may be in its critical region.
- ② *Liveness*: When a process requests access to its critical region, it is granted eventually.
- ③ *No blocking*: A process may request access to its critical region at any time.
- ④ *Flexibility*: Processes need not enter their critical regions in a fixed order (this excludes rigid plans such as cycles).

Mutual Exclusion: Modelling

Each process cycles through the sequence

$$n \rightarrow t \rightarrow c \rightarrow n \rightarrow \dots,$$

where

n – “non critical”

t – “critical region requested” (*trying*)

c – “in the critical region”

Set of states for a system of two processes: states are labelled with elements of a subset of the cartesian product $\{n_1, t_1, c_1\} \times \{n_2, t_2, c_2\}$

Initial state: with label (n_1, n_2)

State transitions: *one* of the processes performs a transition according to the indicated cycle; e.g.

$$(n_1, t_2) \rightarrow (n_1, c_2)$$

Mutual Exclusion: Formal Requirements

Formalisation of the required properties:

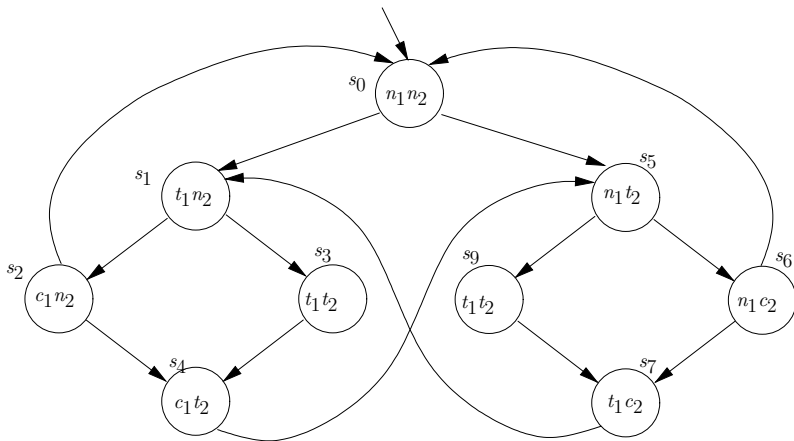
Safety: $\phi_1 := \mathbf{AG} \neg(c_1 \wedge c_2)$

Liveness: $\phi_2 := \mathbf{AG} (t_1 \Rightarrow \mathbf{AF} c_1)$

No blocking: $\phi_3 := \mathbf{AG} (n_1 \Rightarrow \mathbf{EX} t_1)$

Flexibility: $\phi_4 := \mathbf{EF} (c_1 \wedge \mathbf{E} [c_1 \mathbf{U} (\neg c_1 \wedge \mathbf{E} [\neg c_2 \mathbf{U} c_1])])$

Mutual Exclusion: Transition Diagram



Model Checking for CTL

The problem: how can

$$\mathcal{M}, s \models \psi \quad (*)$$

be checked algorithmically?

More generally: find *one* s or *all* s such that $(*)$ holds.

The approach presented here addresses the more general problem.

Main idea:

- Develop incrementally a labelling of the states in \mathcal{M} with those subformulas of ψ that hold in the respective states.
- starting with the atomic formulas,
- until the full formula ψ is reached.
- Given a transition structure $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula ϕ
- Transform ϕ into a form that contains only the reduced set of connectives $\perp, \wedge, \neg, \mathbf{EX}, \mathbf{AF}, \mathbf{EU}$.

Algorithmus for Model Checking

Label incrementally states with subformulas of ϕ according to the structure of formulas.

ψ a subformula of ϕ ;

Assume that labelling of states with the immediate subformulas of ψ is complete.

$\psi := \perp$: no state is to be labelled with \perp .

$\psi := p$ (atomic): s is labelled with p if $p \in L(s)$

$\psi := \psi_1 \wedge \psi_2$: s is labelled with ψ if s is already labelled with ψ_1 as well as ψ_2 .

$\psi := \neg\psi_1$: s is labelled with ψ if s is *not* already labelled with ψ_1 .

$\psi := \mathbf{EX} \psi_1$: s is labelled with ψ if one of its successor states is labelled with ψ_1 .

Algorithmus for Model Checking (2)

$\psi := \mathbf{AF} \psi_1$:

- Every state s that is labelled with ψ_1 is to be labelled with ψ .
- Label s with $\mathbf{AF} \psi_1$ when all successor states of s are labelled with $\mathbf{AF} \psi_1$.
- Repeat this process until no change in labelling occurs.

$\psi := \mathbf{E} [\psi_1 \mathbf{U} \psi_2]$:

- Every s labelled with ψ_2 is to be labelled with ψ .
- s is labelled with ψ if s is labelled with ψ_1 and at least one successor state is labelled with $\mathbf{E} [\psi_1 \mathbf{U} \psi_2]$.
- Repeat this process until no further change in labelling occurs.

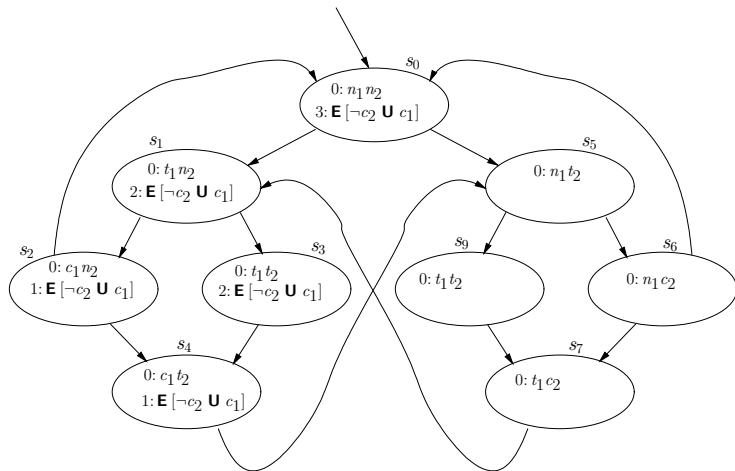
Algorithmus for Model Checking (3)

When the labelling process for ϕ is terminated all states at which ϕ holds have been identified.

The labelling process can be expressed more formally by fixed point computations of state sets derived from the fixed point characterization of the temporal operators.

Example: Mutual Exclusion

Labelling of states for $\mathbf{E} [\neg c_2 \mathbf{U} c_1]$



Symbolic Model Checking with OBDDs

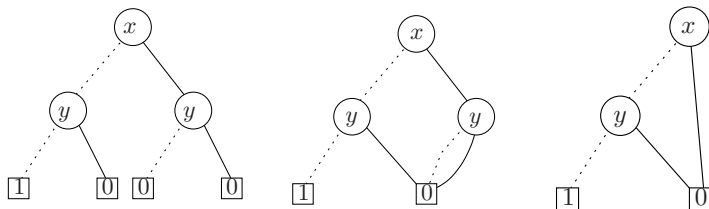
OBDD – “*Ordered Binary Decision Diagrams*”

Binary decision diagram derived from binary *decision tree* with inner nodes labelled by a propositional decision symbol x

Decision symbols follow a (total) order along each path from root to leaf node

- OBDD-based methods are used as efficient decision procedures for validity of propositional formulas
- originally developed for hardware verification
- often orders of magnitude faster than other decision methods
- used in many implementations of model checking

Reducing BDDs



dashed line: 0 (= **F**)

solid line: 1 (= **W**)

1. step: merging of leaf nodes with identical labels
2. step: elimination of nodes without a decision

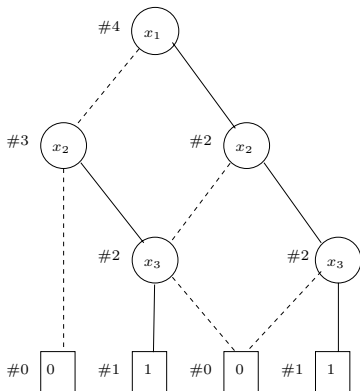
Reduction of OBDDs: *reduce*

Systematic reduction of OBDDs by labelling nodes of the OBDD, e.g. with natural numbers in sequence

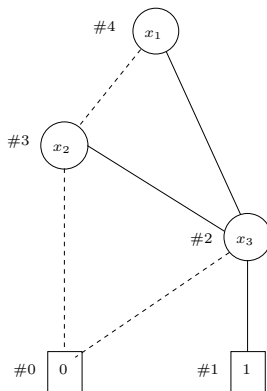
- Leaf nodes are labelled with 0 or 1.
- Starting from the leaf nodes, inner nodes are labelled level by level.
- If the 0- and 1-successors of a n carry the same label, node n gets the same label.
- If there is another node m for the same variable such that its successor nodes carry the same labels as the successors of n , node n receives the label of m .
- Otherwise n receives a new label.

When labelling is completed, all nodes with the same label are merged, and edges are modified accordingly.

reduce – Example



Reduce

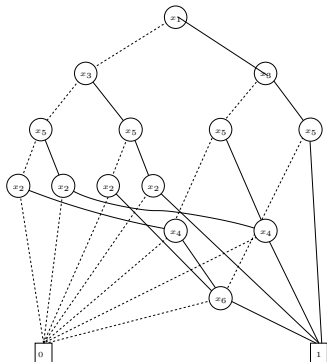
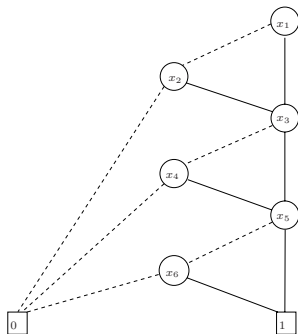


Effects of the Variable Order

OBDD for the formula $(x_1 + x_2) \cdot (x_3 + x_4) \cdot (x_5 + x_6)$ mit Variable order:

(a) $(x_1, x_2, x_3, x_4, x_5, x_6)$,

(b) $(x_1, x_3, x_5, x_2, x_4, x_6)$



Operations on OBDDs

In the following we use an abbreviating notation:

0 and 1 for **F** and **W**, respectively;

\cdot and $+$ for \wedge and \vee , respectively

\bar{x} for $\neg x$

B_f denotes the OBDD for the boolean formula f etc.

op is an arbitrary binary boolean operation.

$f[y/x]$ denotes the formula resulting from replacing any occurrence of x by y (substitution).

Required auxiliary operations:

apply, restrict, exists

OBDD Operations: *apply*

$apply(op, B_f, B_g)$ – computes the reduced OBDD for formula $f \text{ op } g$
assuming the OBDDs B_f, B_g for f and g are already available

The process build on the *Shannon expansion*:

$$f \Leftrightarrow x \cdot f[1/x] + \bar{x} \cdot f[0/x]$$

For *apply*:

$$f \text{ op } g \Leftrightarrow x \cdot (f[1/x] \text{ op } g[1/x]) + \bar{x} \cdot (f[0/x] \text{ op } g[0/x])$$

OBDD Operations: *apply* (2)

Case split according to variables v_f and v_g labelling the root nodes of B_f and B_g , respectively:

(a) $v_f = v_g$: $B_{f \text{ op } g} = \text{if}(v_f, \text{apply}(op, t_f, t_g), \text{apply}(op, f_f, f_g))$

\rightsquigarrow recursive propagation of op into the substructures

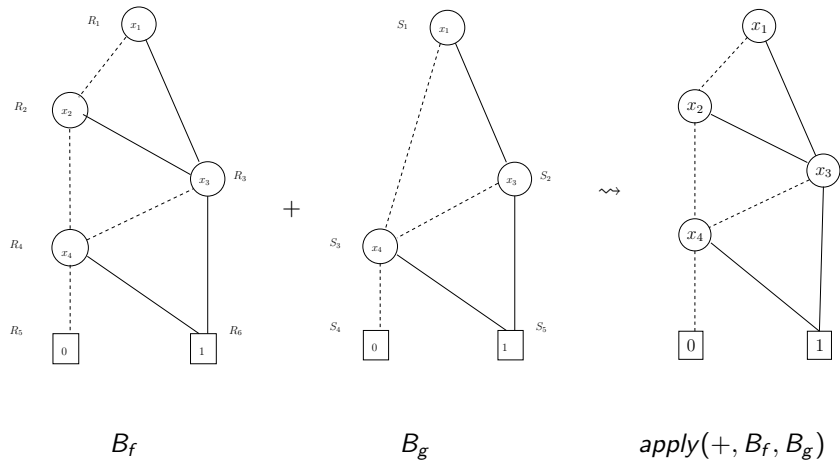
(b) $v_f < v_g$: $B_{f \text{ op } g} = \text{if}(v_f, \text{apply}(op, t_f, B_g), \text{apply}(op, f_f, B_g))$

Due to the ordering of variables, v_A cannot occur in B_g ; hence it suffices to modify the substructures.

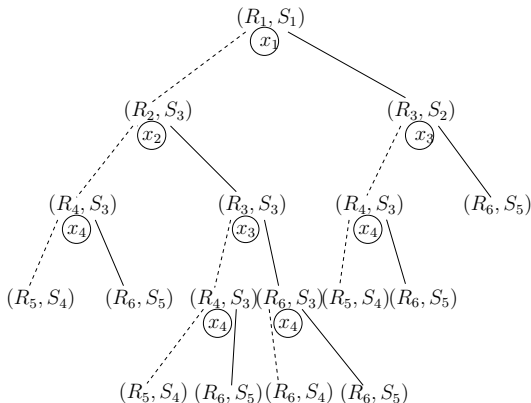
(c) $v_g < v_f$: in analogy to (b)

In general it is necessary to reduce the result subsequently.

apply – Example



apply – Example (2)



Recursive call structure of *apply* for the example *apply*(+, B_f , B_g)

OBDD Operations: *restrict*

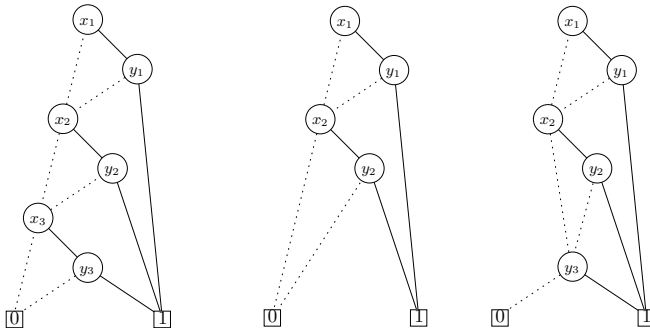
$restrict(b, x, B_f)$ computes a reduced OBDD for $f[b/x]$ using the same variable order as B_f ($b \in \{0, 1\}$)

For each node labelled with x all incoming edges are re-linked to the respective successor node; the node itself is removed.

The resulting BDD needs to be reduced.

restrict – Example

BDDs for $f := x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$ and restrictions on x_3



OBDD Operations: *exists*

$exists(x, B_f)$ computes a reduced OBDD for the existentially quantified formula $\exists x.f$

$$\exists x.f := f[0/x] + f[1/x]$$

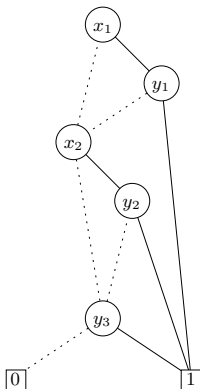
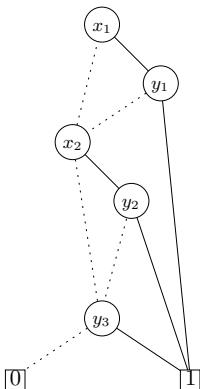
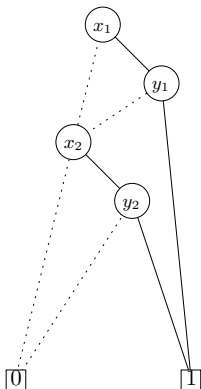
exists may be implemented by

$$exists(x, B_f) = apply(+, restrict(0, x, B_f), restrict(1, x, B_f))$$

exists – Example

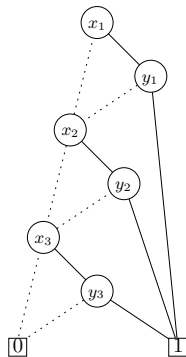
f as before;

$B_1 = \text{restrict}(0, x_3, B_f)$, $B_2 = \text{restrict}(1, x_3, B_f)$, $\text{apply}(+, B_1, B_2)$

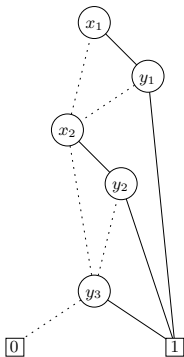


exists – Example (2)

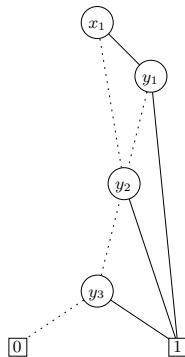
f as before



$\exists x_3.f$



$\exists x_2.\exists x_3.f$



Model Checking with OBDDs

Basic idea: Encode all entities involved in model checking by propositional formulas and use an OBDD representation for further processing

- Representation of the state transition system:
 - Finite state sets are encoded as vectors of boolean values.
 - A transition is encoded as conjunction of the encoding of start and target state
- CTL formulas are encoded as (representations of) state sets.
- Checking is performed by construction and comparison of OBDDs.

Model Checking with OBDDs (2)

Representation of finite state sets and subsets:

- Choose a “sufficiently large” binary vector $\{0, 1\}^n$ to represent states
- A subset $T \subseteq S$ is represented by its characteristic funktion

$$f_T : \{0, 1\}^n \rightarrow \{0, 1\}$$

- The set operations intersection, union and complement are represented by means of the corresponding logical operations \wedge , \vee and \neg .

Model Checking with OBDDs (3)

For encoding a CTL model $\mathcal{M} = (S, \rightarrow, L)$ the labelling function L may be used:

- Assume a fixed ordering of atoms.
- Every state is encoded by the conjunction of the positive and negative values of the atoms at that state.
- Different states have to have different encodings; atoms may be added as needed.
- A subset of states is represented by the disjunction of the (encoding of the) states belonging to the subset.

Model Checking with OBDDs (4)

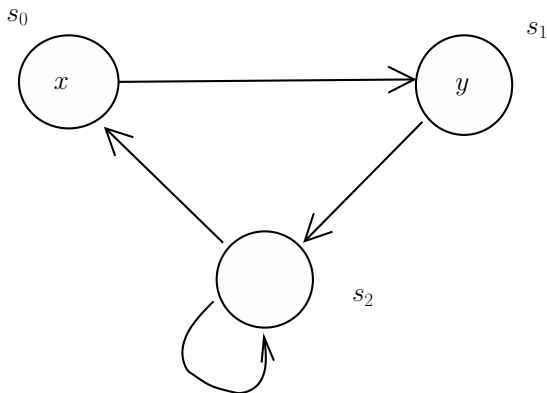
Representation of the state transition relation:

- Transition relation is a subset of $S \times S$
 - \rightsquigarrow 2 copies of the representation of S are required
 - \rightsquigarrow a primed copy a' for each atom a
- Transition $s \rightarrow s'$ is represented by the conjunction of the representations of s (using a -s) and s' (using a' -s)
- The full transition relation is represented as disjunction of all individual transitions.

For the computation of predecessors of states it is necessary to rename atoms appropriately (a to a' etc.)

Model Checking with OBDDs: Example

State transition diagram:



Model Checking with OBDDs: Example (2)

$$S := \{s_0, s_1, s_2\}$$

$$\rightarrow := \{(s_0, s_1), (s_1, s_2), (s_2, s_0), (s_2, s_2)\}$$

$$L(s_0) := \{x\} \quad L(s_1) := \{y\} \quad L(s_2) := \emptyset$$

Encoding of state sets:

Set	Boolean values	Boolean function
\emptyset		\perp
$\{s_0\}$	$(1, 0)$	$x \cdot \bar{y}$
$\{s_1\}$	$(0, 1)$	$\bar{x} \cdot y$
$\{s_2\}$	$(0, 0)$	$\bar{x} \cdot \bar{y}$
$\{s_0, s_1\}$	$(1, 0), (0, 1)$	$x \cdot \bar{y} + \bar{x} \cdot y$
$\{s_0, s_2\}$	$(1, 0), (0, 0)$	$x \cdot \bar{y} + \bar{x} \cdot \bar{y}$
$\{s_1, s_2\}$	$(0, 1), (0, 0)$	$\bar{x} \cdot y + \bar{x} \cdot \bar{y}$
S	$(1, 0), (0, 1), (0, 0)$	$x \cdot \bar{y} + \bar{x} \cdot y + \bar{x} \cdot \bar{y}$

Model Checking with OBDDs: Example (3)

Encoding of the transition relation:

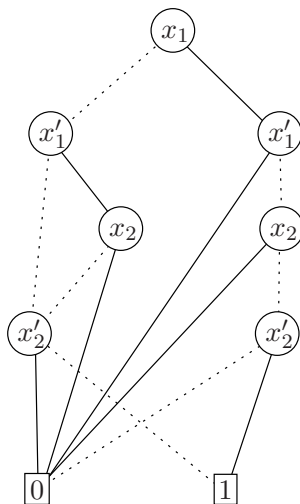
- Each individual transition represented by conjunction of codes for start and target states
- Target state always using primed copies of variables
- Full transition relation represented by disjunction of codes for individual transitions

$$f^{\rightarrow} := x \cdot \bar{y} \cdot \bar{x}' \cdot y' + \bar{x} \cdot y \cdot \bar{x}' \cdot \bar{y}' + \bar{x} \cdot \bar{y} \cdot x' \cdot \bar{y}' + \bar{x} \cdot \bar{y} \cdot \bar{x}' \cdot y'$$

For the representation by OBDDs it is recommended to choose an order that puts variables x and x' in sequence, such as $\{x, x', y, y'\}$.

Model Checking with OBDDs: Example (4)

OBDD for the transition relation $f \rightarrow$ in the example:



Model Checking: Required Operations

- Union, intersection, and complement (\setminus)
- Computation of the set of predecessors of a set X (in SAT_{EX} , SAT_{EU}):

$$\text{pre}_{\exists}(X) := \{s \in S \mid \text{there exists } s' \in X \text{ mit } s \rightarrow s'\}$$

- Restricted all-quantified predecessor set of a set X (in SAT_{AF}):

$$\text{pre}_{\forall}(X) := \{s \in S \mid s' \in X \text{ for alle } s' \in S \text{ such that } s \rightarrow s'\}$$

$$\text{pre}_{\forall} \text{ can be reduced to } \text{pre}_{\exists} \text{ by } \text{pre}_{\forall}(X) = S \setminus \text{pre}_{\exists}(S \setminus X)$$

Realisation of pre_{\exists} :

$$\text{pre}_{\exists}(X) := \{s \in S \mid \text{there exists } s' \in X \text{ such that } s \rightarrow s'\}$$

- In $\text{pre}_{\exists}(X)$ the variable X denotes a set of successor states
 \rightsquigarrow a copy $B_{X'}$ of B_X using primed variables x' etc. needed
- Using the operations *apply* and *exists* one computes $\text{pre}_{\exists}(X)$ by
 $\text{exists}(\hat{x}', \text{apply}(\cdot, B_{\rightarrow}, B_{X'}))$

\hat{x}' denotes the vektor of variables x' which encode states.

Linear Time Logic

LTL: “*Linear Time Logic*” differs from CTL in that the logic builds on a *linear* notion of time.

- An LTL formula expresses a property of a single path; a path corresponds to one (linear) time line.
There is no branching of time as in CTL.
- There no explicit path quantors corresponding to **A** and **E**.
In particular, it is not possible to express the *existence* of a path with a certain property.
However: one often considers the set of *all* paths starting at a particular state (a kind of implicit universal quantification).
- It is possible to nest the operators **F**, **G**, and **X**.

LTL: Syntax and Semantics

Syntax of LTL formulas:

$$\phi ::= \perp \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid \dots \mid \\ \mathbf{X}\phi \mid \mathbf{G}\phi \mid \mathbf{FF}\phi \mid \phi \mathbf{U}\phi$$

p an atomic formula

The *semantics* of LTL formulas is defined relative to a path (or set of paths).

For a path $\pi = s_1, s_2, \dots$ in a state transition system $\mathcal{M} = (S, \rightarrow, L)$ denotes $\pi^i = s_i, s_{i+1}, \dots$ a suffix of π starting at state s_i .

LTL: Semantics

$\mathcal{M}, \pi \models \phi$ path π (in model \mathcal{M}) satisfies the formula ϕ
as usual defined recursively over the structure of LTL formulas

$\mathcal{M}, \pi \models \phi$ for propositional ϕ obvious

$\mathcal{M}, \pi \models \mathbf{X} \phi$ iff $\mathcal{M}, \pi^2 \models \phi$

$\mathcal{M}, \pi \models \mathbf{G} \phi$ iff $\mathcal{M}, \pi^i \models \phi$ for every $i \geq 1$.

$\mathcal{M}, \pi \models \mathbf{FF} \phi$ iff there exists a $i \geq 1$ such that $\mathcal{M}, \pi^i \models \phi$

$\mathcal{M}, \pi \models \phi_1 \mathbf{U} \phi_2$ iff there exists a $i \geq 1$ such that $\mathcal{M}, \pi^i \models \phi_2$
and $\mathcal{M}, \pi^j \models \phi_1$ for every $1 \leq j < i$

LTL Equivalences

Two LTL formulas ϕ and ψ are *semantically equivalent* if they are satisfied by the same paths.

Examples of LTL equivalences:

$$\mathbf{G} \phi \Leftrightarrow \neg \mathbf{FF} \neg \phi$$

$$\mathbf{FF}(\phi \vee \psi) \Leftrightarrow \mathbf{FF} \phi \vee \mathbf{FF} \psi$$

$$\mathbf{G}(\phi \wedge \psi) \Leftrightarrow \mathbf{G} \phi \wedge \mathbf{G} \psi$$

$$\phi \mathbf{U} \psi \Leftrightarrow \neg(\neg \psi \mathbf{U} (\neg \phi \wedge \neg \psi)) \wedge \mathbf{FF} \psi$$

$$\mathbf{FF} \phi \Leftrightarrow \top \mathbf{U} \phi$$

$$\phi \mathbf{W} \psi \Leftrightarrow (\phi \mathbf{U} \psi) \vee \mathbf{G} \phi$$

Comparison of CTL and LTL

CTL and LTL formulae may express the same properties in different ways:
example:

“Every p is eventually followed by q ”

in CTL: **AG** ($p \Rightarrow \mathbf{AF} q$) in LTL: **G** ($p \Rightarrow \mathbf{FF} q$)

However, there exist CTL formulas for which no equivalent LTL formulas exist, and vice versa. Examples:

AG (**EF** p) – “it is always possible to reach p ” (CTL)

A [**G** $\mathbf{FF} p \Rightarrow \mathbf{FF} q$]

“If p occurs infinitely often along a path, then q also occurs” (LTL)

An application of the latter is a fairness condition: A request that is repeated often enough (i.e. infinitely often) will eventually be served.

Model Checking for LTL

The model checking problem for LTL:

For a given model (state transition system) $\mathcal{M} = (S, \rightarrow, L)$ and an LTL formula ϕ ,

Do all all paths of \mathcal{M} starting at state s satisfy ϕ ?

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

The CTL approach to model checking does not transfer to LTL; LTL formulas are evaluated along a path, not for a state or a set of states.

Model Checking for LTL (2)

Basic approach to model checking of LTL formulas:

- Construct an automaton (state transition system) $A_{\neg\phi}$ for $\neg\phi$.
The automaton characterizes exactly those paths which satisfy $\neg\phi$.
- The constructed automaton $A_{\neg\phi}$ is combined with the model \mathcal{M} .
Result is a transition system that contains paths that are paths of both the automaton and the model.
- Check whether there is path in the combined system starting at s .
If no such path exists, \mathcal{M} satisfies formula ϕ at s .
Otherwise the found path indicates a counter-example.

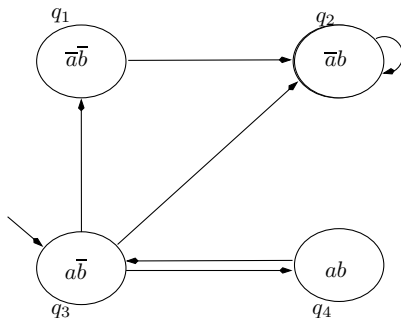
Example

The model for which the formula

$$\phi := \neg(a \text{ U } b)$$

is to be checked.

(The path q_3, q_2, q_2, \dots
does not satisfy ϕ .)

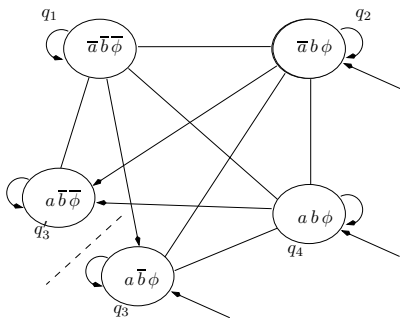


Example (2)

Automaton $A_{a \mathbf{U} b}$ for accepting exactly those paths that satisfy $a \mathbf{U} b$.

Edges without arrows denote transitions in both direction.

Acceptance condition: Path must not include an infinite loop through state q_3 .



Example (3)

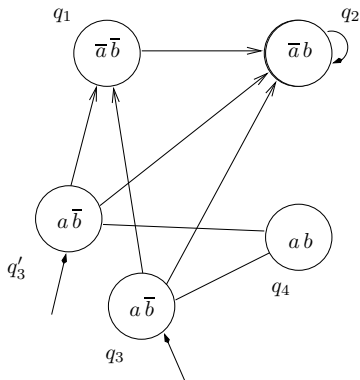
Remarks on the automaton:

- The states cover all possible combinations of values of the atoms a and b .
- There exist two states for $a\bar{b}$, depending on whether or not $a \mathbf{U} b$ is satisfied (for all other states the values a and b are sufficient).
- Starting at q_3 , only transitions to states that satisfy b are possible.
- Otherwise, transitions from any state to any other state are possible.

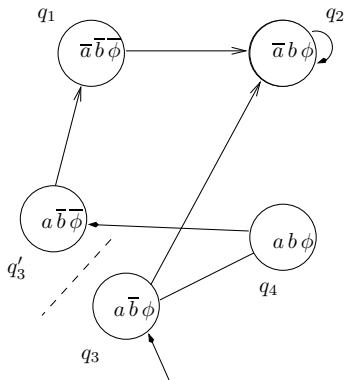
Combination of model and automaton by superposition: only those transitions remain that are possible in both structures.

For this purpose: Adaptation of the model by splitting state q_3 .

Example (4)



Modified model



Combined transition system

Path in the combined system satisfying the acceptance condition:

q_3, q_2, q_2, \dots

Limitations

- *State explosion*: The number of states grows exponentially with the number of propositional variables.

Even though the model checking techniques – in particular those using BDDs – are capable of handling models with a very large sets of states (such as 10^{30} states and more), application problems eventually reach the limit of what can be dealt with in practice.

Methods of trying to overcome the limit by reducing state spaces include

- Exploitation of symmetries
- Simplifying models by abstraction
- *Infinite state spaces*: The model checking techniques presented here require the state space to be finite. Infinite state spaces occur naturally in applications (e.g. by including variables with unbounded integer values).

Again, abstraction techniques may be useful in trying to reduce infinite state spaces to finite ones.

Part II

Theorem Proving

Friedrich v. Henke and Holger Pfeifer

Part Outline

- ① Introduction
- ② Logical Foundations
- ③ PVS Specifications and Proofs

Introduction

Theorem Proving

- **Theorem Proving:** demonstration of the truth of some statement (“theorem”) through logical deduction
- **Mechanized Theorem Proving:** to prove theorems by a computer program
- process of derivation according to rules of some **formal logic**, such as first-order logic
- the logic defines what is considered a “proof”
- mechanisation of theorem proving to ensure that
 - both the process and the result, i. e. the proof ...
 - ... are comprehensible, checkable, and repeatable

Theorem Proving Applications

Main application areas of theorem proving:

- correctness of **hardware designs**
- modelling and formal analysis of **critical systems**
 - safety, dependability, trustworthiness, etc., of systems and system components
- theorem prover becomes an essential inference component in the practical deployment of formal methods
- prover modules as auxiliary components for inference tasks – **“embedded intelligence”**
- **proof-carrying code**
- typically, emphasis is on the combination of methods, rather than deductions in a pure calculus

Proofs and Automation (1)

- **Proof Checking:** computer is presented a proof and checks whether it adheres to the rules of the logic
- **Automatic Theorem Proving:** computer program tries to construct a proof autonomously, without the aid of a human
- **Interactive Theorem Proving:** computer program serves as a proof assistant to a human user
 - routine tasks are carried out by the computer
 - user provides manual control over the global construction of a proof
- **Tactical Theorem Proving:** use of “tactics” or “strategies” for proof search, programmed construction of proofs

The highest possible degree of automation is (in principle) desirable; however, ...

Proofs and Automation (2)

... fully automatic theorem proving is inherently difficult, resp. in principle impossible.

- Sufficiently complex logics (such as first-order logic) are undecidable
- Proof procedures even for decidable logics and theories (e. g. propositional logic, linear arithmetic) exhibit high complexity (NP-complete, PSPACE, ...)
- In principle: the simpler the logic, the higher the chance for fully automatic proofs in practice.
- e. g., certain classes of propositional problems can be solved efficiently with new approaches
 - binary decision diagrams
 - SAT-solver, SMT-solver
- On the other hand, a highly-expressive – and therefore complex – logic is desirable for complex applications

Logic vs. Specification Language

Practical use of mechanized theorem proving for verification requires an expressive language for modelling the application at hand.

Expressiveness of the language of pure logics – of any kind – is still too weak.

- mathematical practice: proofs are carried out within certain theories, e. g. group theory, as characterized through group axioms
- theories need a syntactical framework
 - for structuring, for the composition of theories, for readability
- Typing: as helpful in mechanized theorem proving as for programming languages (for essentially the same reasons)
- support to build and maintain libraries of theories and theorems
- Modelling is facilitated by a highly expressive language

Therefore: use of a dedicated **specification language**

- based on an expressive logic, and
- enriched by various constructs similar to those found in programming languages

PVS – Prototype Verification System (1)

PVS is a verification system combining a very expressive specification language and a powerful prover component.

- **interactive** theorem prover
- developed at SRI International
- PVS provides:
 - higher-order logic as the logical foundation
 - expressive specification language
 - complex type system
 - powerful inference machinery, including decision procedures
 - simple tactic language for definition of new proof strategies
 - extensive prelude and library containing numerous specifications and proved facts

PVS – Prototype Verification System (2)

There are several similar systems, e. g.

- Isabelle, L. Paulson (Cambridge), T. Nipkow (Munich)
- HOL (“Higher-Order Logic”), R. Milner (Stanford), M. Gordon (Cambridge), T. Melham (now Oxford) – several variants
- ACL 2 (“A Computational Logic”), M. Kaufmann, J S. Moore (Austin, Texas)
- Coq, G. Huet (INRIA-Rocquencourt), T. Coquand (now Chalmers),

PVS is used in this course because

- PVS provides adequate capabilities
- alternative systems do have their strengths, but are not obviously “better”
- PVS is accepted and used world-wide

Logical Foundations

- Typed First-Order Logic
- Sequent Calculus
- Lambda Calculus
- Higher-Order Logic
- Induction, Recursion and ADTs

Prerequisites (such as provided by the course *Logic in Computer Science*):

- Basics of Propositional Logic
- Basics of First-Order Predicate Logic

Typed First-Order Logic

Classical First-Order Logic is untyped. **Typed FOL** (as used as a basis of PVS) extends the usual untyped logic to allow different types, or sorts, of values:

- different individuals can be assigned different kinds of values
- in addition to their arity, function symbols and predicate symbols now also come with a **signature** to define the types of the respective arguments and results
- Predefined sorts include at least a type *Bool* of truth values.
- Notation for typing:
 $t : S$ means “term t has type S ”

Syntax

Typing rules

- When constructing terms, the typing rules must be obeyed:

$$\frac{f : S_1 \times S_2 \times \dots \times S_n \rightarrow S_{n+1} \quad t_1 : S_1 \quad \dots \quad t_n : S_n}{f(t_1, \dots, t_n) : S_{n+1}}$$

- Analogously for predicates
- Quantifiers are typed, too:

$$\forall v : S. P(v)$$

$$\exists v : S. P(v)$$

for $P : S \rightarrow \text{Bool}$, for each $S \in \mathcal{S}$

- Other notions remain the same (ground term, bound variable, scope, etc.)

Semantics of Typed FOL

Structures are \mathcal{S} -indexed: $\mathcal{A}_{\mathcal{S}} = (\mathcal{U}_{\mathcal{S}}, \mathcal{I}_{\mathcal{S}})$

- ① **Universe** $\mathcal{U}_{\mathcal{S}}$ consists of **non-empty** sets M_S for each type $S \in \mathcal{S}$, in particular:

$$M_{Bool} := \{true, false\}$$

- ② **Interpretation** $\mathcal{I}_{\mathcal{S}}$ maps
 - variables $v \in V_{\mathcal{S}}$ to objects

$$\mathcal{I}_{\mathcal{S}}(v) \in M_S \text{ for each } S \in \mathcal{S}$$

- function symbols $f : S_1 \times S_2 \times \dots \times S_n \rightarrow S_{n+1}$ to n -ary functions

$$\mathcal{I}_{\mathcal{S}}(f) : M_{S_1} \times M_{S_2} \times \dots \times M_{S_n} \rightarrow M_{S_{n+1}}$$

- predicate symbols $P : S_1 \times S_2 \times \dots \times S_n \rightarrow Bool$ to relations

$$\mathcal{I}_{\mathcal{S}}(P) : M_{S_1} \times M_{S_2} \times \dots \times M_{S_n} \rightarrow M_{Bool}$$

or, equivalently,

$$\mathcal{I}_{\mathcal{S}}(P) \subseteq M_{S_1} \times M_{S_2} \times \dots \times M_{S_n}$$

Proof methods

For first-order logic, a variety of proof methods exist, for example:

- rewriting with equivalences
 - prove a formula true by rewriting it to **T**
- resolution methods
 - refutation method: prove a formula by deriving a contradiction from its negation
 - involves representing the formula as sets of “clauses” where quantifiers are eliminated
 - successive “cancellation” of complementary literals **L** and $\neg L$ until empty set is derived
- tableaux methods
 - refutation method: start from the negation of formula to prove
 - rules describe how to decompose formula
 - tableaux is a directed graph, nodes are sub-formulas
 - formula is proved, if there is a contradiction on every path
- sequent calculus: this is the basis for the PVS prover
- ...

Sequent Calculus

The sequent calculus is a deduction system for first-order logic; it is the basis of the PVS prover.

- The basic entity of the calculus are **sequents**: $\Gamma \vdash \Delta$
- Γ and Δ are finite sets (or multisets) of formulae
- Γ is called the **antecedent**, and Δ is the **succedent**
- Intuitive interpretation: a sequent $A_1, \dots, A_m \vdash B_1, \dots, B_n$ corresponds to the statement that the formula $A_1 \wedge \dots \wedge A_m \Rightarrow B_1 \vee \dots \vee B_n$ is true
- Derivation rules describe how to derive new sequents from given ones
- Practically, deriving a sequent $\Gamma \vdash \Delta$ means to prove some formula out of the set of formulae Δ from a set of preconditions Γ

Sequent Calculus

There is only one axiom:

$$\Gamma, A \vdash A, \Delta$$

Structural rules:

- Contraction

$$\frac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta} \text{ contr } L$$

$$\frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A} \text{ contr } R$$

- Cut-rule

$$\frac{\Gamma \vdash \Delta, A \quad A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} \text{ cut}$$

Sequent Calculus: Propositional Rules

- Implication

$$\frac{\Gamma \vdash \Delta, A \quad B, \Gamma \vdash \Delta}{A \Rightarrow B, \Gamma \vdash \Delta} \Rightarrow L$$

$$\frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash A \Rightarrow B, \Delta} \Rightarrow R$$

- Conjunction

$$\frac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \wedge L$$

$$\frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B} \wedge R$$

- Disjunction

$$\frac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta} \vee L$$

$$\frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B} \vee R$$

- Negation

$$\frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta} \neg L$$

$$\frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A} \neg R$$

Sequent Calculus: Quantifier Rules

- Universal quantifier:

$$\frac{A[x \leftarrow t], \Gamma \vdash \Delta}{\forall x. A, \Gamma \vdash \Delta} \forall L$$

$$\frac{\Gamma \vdash \Delta, A[x \leftarrow y]}{\Gamma \vdash \forall x. A, \Delta} \forall R$$

- Existential quantifier:

$$\frac{A[x \leftarrow y], \Gamma \vdash \Delta}{\exists x. A, \Gamma \vdash \Delta} \exists L$$

$$\frac{\Gamma \vdash \Delta, A[x \leftarrow t]}{\Gamma \vdash \exists x. A, \Delta} \exists R$$

Restrictions:

- **Variable Capture:** t must be *free for* x in A .
Free variables in t must not be captured by a quantifier in A
- **Eigenvariable:** y is a *fresh* variable.
Variable y must not occur free in the succedent of $\forall R$ and $\exists L$.

Proofs in the sequent calculus

Proofs are constructed **backwards**:

- start with the sequent to be proved
- successively apply the deduction rules
- until all branches reach an instance of the axiom

Example: prove $(P \Rightarrow Q) \wedge (Q \Rightarrow R) \wedge \neg R \vdash \neg P$

$$\frac{\frac{\frac{Q \Rightarrow R, P \vdash R, P}{P \Rightarrow Q, Q \Rightarrow R, P \vdash R} \Rightarrow L \quad \frac{\frac{Q, P \vdash Q, R \quad Q, R, P \vdash R}{Q, Q \Rightarrow R, P \vdash R} \Rightarrow L}{P \Rightarrow Q, Q \Rightarrow R, \neg R \vdash \neg P} \neg L \text{ and } \neg R}{(P \Rightarrow Q) \wedge (Q \Rightarrow R) \wedge \neg R \vdash \neg P} \wedge L \text{ (2 times)}$$

Observations

- Antecedent and succedent are treated **symmetrically**
- Applicability of a deduction rule determined solely by **structure** of formula
- Rules have a **sub-formula property**:
every formula that occurs in the antecedent of a rule also occurs as a (sub-)formula in the succedent

Exception: *cut*-rule

- No need to “guess” a formula when applying rules backwards
- *Cut*-elimination: every proof that contains an application of the *cut*-rule can be transformed into one without.
However: length of proof can grow over-exponentially
- Rule set (without *cut*-rule) provides **decision procedure** for propositional logic

Proof Example with Quantifiers

Prove: $\exists x. \forall y. P(x, y) \vdash \forall y. \exists x. P(x, y)$

$$\frac{\frac{\frac{P(v, w) \vdash P(v, w)}{P(v, w) \vdash \exists x. P(x, w)} \exists R}{\forall y. P(v, y) \vdash \exists x. P(x, w)} \forall L}{\forall y. P(v, y) \vdash \forall y. \exists x. P(x, y)} \forall R}{\exists x. \forall y. P(x, y) \vdash \forall y. \exists x. P(x, y)} \exists L$$

Lambda Calculus: λ -expressions

λ -expressions: notation to denote **anonymous functions** with a single argument

- instead of writing

$$f(x) = 3 \cdot x + 4$$

we write

$$f = \lambda x. 3 \cdot x + 4$$

- λ -expressions may be used at places where one usually has a function name:

$$(\lambda x. 3 \cdot x + 4) (2 \cdot a)$$

- Functions are first-class objects:
 - can be used as arguments in a function application
 - can be the result value of a function:

$$\lambda x. \lambda y. x + y$$

- **λ -variable** x in $\lambda x. e$ corresponds to formal parameter of function definitions in programming languages

Definition (λ -Term)

The set of λ -terms is inductively defined by the following three clauses:

- 1 all variable symbols x, y, \dots are λ -terms
 - 2 λ -abstraction: if x is a variable and e is a λ -term, then $\lambda x. e$ is a λ -term
 - 3 λ -application: if f and e are λ -terms, then so is $f(e)$.
- λ binds variable x in e . Concepts of bound/free variables, scope, etc. analogously to first-order logic
 - notation varies: *juxtaposition* for application: ffx for $f(f(x))$
 - relaxed notation:
 - combined binding of several variables: $\lambda x, y. e$ instead of $\lambda x. \lambda y. e$
 - infix notation for certain standard function symbols, e. g. $x + 1$ instead of $+(x)(1)$
 - parentheses are used when necessary

Conversions

Calculating with λ -expressions:

- α -conversion: renaming of bound variables

$$(\lambda x. e) = (\lambda y. e[x \leftarrow y])$$

provided that y does not occur in e .

- β -conversion: evaluation of function application

$$(\lambda x. e_1)(e_2) = e_1[x \leftarrow e_2]$$

provided that e_2 is free for x in e_1 : free variables of e_2 must not get bound by a λ of e_1 .

“Capturing” of variables can always be avoided by adequate renamings of variables in e_1 through α -conversion.

Example: β -conversion

$$\begin{aligned} & (\lambda x. (\lambda f. f(f(x))) (\lambda x. x * x + 1)) (2) \\ &= \dots \\ &= ((2 * 2 + 1) * (2 * 2 + 1)) + 1 \\ &= \dots = 26 \end{aligned}$$

Observe:

- **order of reductions** depends on evaluation strategy: *call-by-name* and *call-by-value*, *innermost*, *outermost*, etc.
- call-by-name can yield to multiple evaluation of same term, but can save unneeded evaluations: **lazy evaluation**

λ -calculus

The λ -calculus is a formal calculus to derive equalities between λ -terms.

- Axioms:

- α -conversion: $(\lambda x. e) = (\lambda y. e[x \leftarrow y])$
- β -conversion: $(\lambda x. e_1)(e_2) = e_1[x \leftarrow e_2]$
- reflexivity: $M = M$

- Rules:

$$\frac{M = N}{N = M}$$

$$\frac{M = N \quad N = L}{M = L}$$

$$\frac{L = K \quad M = N}{L(M) = K(N)}$$

$$\frac{M = N}{\lambda x. M = \lambda x. N} \quad \xi$$

β -conversion leads to a *reduction relation* (β -reduction) among λ terms, which is the core of λ -calculus

Extensionality

Definition (Extensionality)

Two functions f and g are **extensionally equal**, if they agree on all inputs:

$$(\forall x. f(x) = g(x)) \Rightarrow f = g$$

with x not occurring free in f or g .

- Only the values are important, not the way they are calculated
- Extensionality rule can be added to the λ -calculus
- Extensionality implies so-called **η -conversion**:

$$\lambda x. f(x) = f \quad (x \text{ not free in } f)$$

- Alternatively, one can add η -rule to the calculus.

Simply-typed λ -calculus

- λ -calculus allows self-application, e. g. $\lambda x. x(x)$
- Typed versions of λ -calculus assign a type to λ -terms
- Notation: $M : T$ denotes that term M has type T
- Augment language with type expressions:
 - type constants, e. g. $Bool$, Nat , ...
 - function types: $S \rightarrow T$
 - product types: $S \times T$
- Variables in λ -bindings are now typed: $\lambda x : T. e$
- Note: product types are not really necessary
 - functions of type $T_1 = (S_1 \times S_2) \rightarrow S_3$ can be transformed into ones of type $T_2 = S_1 \rightarrow (S_2 \rightarrow S_3)$ and vice versa
 - for $f : T_1$, function $\lambda x : S_1. \lambda y : S_2. f(x, y)$ is of type T_2 .
 - for $g : T_2$, function $\lambda(x, y) : (S_1 \times S_2). g(x)(y)$ is of type T_1 .

Transformation from T_1 to T_2 is called **Currying**

Typing rules

- Well-formed λ -terms: those that can be assigned a type
- Typing is relative to a context Γ
- Typing judgement $\Gamma \vdash M : T$ means: “in context Γ , term M has type T ”
- Typing rules

$$\overline{\Gamma, M : T \vdash M : T}$$

$$\frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash (\lambda x : S. e) : S \rightarrow T}$$

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash f : S \rightarrow T}{\Gamma \vdash f(e) : T}$$

- Typing rules disallow terms such as $x(x)$
- Conversion rules are the same as in the untyped λ -calculus

Semantics of the simply-typed λ -calculus

Interpretation of λ terms over a family of non-empty **carrier sets** U_T

Variables x of type T are mapped to elements of the corresponding carrier set: $\mathcal{I}_A(x) \in U_T$

Interpretation of types:

- type constants T are mapped to a carrier set U_T

$$\mathcal{I}_A(\text{Bool}) := \{\text{true}, \text{false}\} \quad \mathcal{I}_A(\text{Nat}) := \{0, 1, 2, \dots\}$$

- function type $S \rightarrow T$ is interpreted by the set of functions mapping elements of $\mathcal{I}_A(S)$ to elements of $\mathcal{I}_A(T)$:

$$\mathcal{I}_A(S \rightarrow T) := \mathcal{I}_A(T)^{\mathcal{I}_A(S)}$$

- product type $S \times T$ is mapped to the Cartesian product of the carrier sets of S and T : $\mathcal{I}_A(S \times T) := \mathcal{I}_A(S) \times \mathcal{I}_A(T)$

Interpretations of terms are values in the respective carrier sets.

Relevance of λ -calculus

- Untyped λ -calculus as an **abstract model** for theory of **computability**, see Church's thesis
- Foundation for most **functional programming languages**, such as LISP, etc.
- Notions of **calculation** and **derivation** coincide
- Foundation for **domain theory**, which is the basis for classical **denotational semantics** of programming languages
- **Extensions** of the simply-typed λ -calculus
 - λ -calculus with polymorphic types
 - type theories: higher-order λ -calculi, which allow abstractions over types
- Foundation for **higher-order logics**

Higher-Order Logic

PVS (like other prover systems like Isabelle, Coq, HOL) builds on **higher-order logic** to gain more expressiveness than is available in FOL. Examples of properties that can not be expressed with first-order formulas:

- the **smallest** model of a set of formulae
- the **transitive closure** of a relation
- generally: the smallest set having a certain property
- first-order formulae can not distinguish finite from infinite structures

Example: characterization of **finite structures** in **2nd-order logic**:

- sets in which all injective mappings are also surjective

$$\forall f. \text{injective}(f) \Rightarrow \text{surjective}(f)$$

$$\text{injective}(f) \quad := \quad \forall x, y. f(x) = f(y) \Rightarrow x = y$$

$$\text{surjective}(f) \quad := \quad \forall y. \exists x. f(x) = y$$

Syntax of Higher-Order Logic vs. FOL

Main syntactical difference between FOL and higher-order logic:

- First-order logic:
 - constants and variables only for **individuals**:
function symbols and predicate symbols denote **function constants** and **predicate constants**
 - quantification over individuals only

$$\forall x. \exists y. P(x, y) \wedge Q(y)$$

- Higher-order logic:
 - variables also for functions and predicates
 - quantification over function and predicate variables

$$\forall v, w. \forall X. \exists Y. \forall f. X(v, w) \wedge Y(f(v))$$

Higher-order logic: syntax and semantics

Syntax of HOL

- **Terms**

- terms are those of the simply-typed λ -calculus

- **Atomic formulae**

- applicative terms of type *Bool*
- example: $M(x)$, where $M : S \rightarrow \text{Bool}$ and $x : S$

- **Well-Formed Formulae**

- constructed as usual from atomic formulae and logical connectives \wedge , \vee , \Rightarrow , \dots
- quantification is allowed over variables of arbitrary type
- example:
$$\forall v, w : S. (\exists X : S \rightarrow \text{Bool}. X(v, w)) \Rightarrow (\exists Y : S \rightarrow \text{Bool}. Y(w, w))$$

Semantics of Higher-Order Logic:

- Combined standard semantics of FOL and the simply-typed λ -calculus

Sequent Calculus for Higher-order Logic

Extension of sequent calculus for FOL with rules for higher-order quantification

- Universal quantifier:

$$\frac{\psi[X \leftarrow \sigma], \Gamma \vdash \Delta}{\forall X : S. \psi, \Gamma \vdash \Delta} \forall L^\omega$$

$$\frac{\Gamma \vdash \Delta, \psi[X \leftarrow Y]}{\Gamma \vdash \Delta, \forall X : S. \psi} \forall R^\omega$$

- Existential quantifier:

$$\frac{\psi[X \leftarrow Y], \Gamma \vdash \Delta}{\exists X : S. \psi, \Gamma \vdash \Delta} \exists L^\omega$$

$$\frac{\Gamma \vdash \Delta, \psi[X \leftarrow \sigma]}{\Gamma \vdash \exists X : S. \psi, \Delta} \exists R^\omega$$

X and Y are predicate variables, and $\sigma := \lambda x_1, \dots, x_n. R(x_1, \dots, x_n)$ a predicate expression of suitable type

Restrictions (cf. rules for FOL):

- No variable capture in $\forall L^\omega$ and $\exists R^\omega$
- In $\forall R^\omega$ and $\exists L^\omega$, variable Y must not occur free in the succedent

Examples

- Modelling sets and set operations
 - correspondence between sets and predicates
 - entities of a type S satisfying a predicate P form a subset of the carrier set of S
 - example: set inclusion $M \subseteq N$

$$\subseteq := \lambda M, N : (S \rightarrow Bool). \forall x : S. M(x) \Rightarrow N(x)$$

- Modelling properties of functions
 - compact formulation of properties
 - example: injectivity of a function

$$injective := \lambda f : (T_1 \rightarrow T_2). \forall x, y : T_1. f(x) = f(y) \Rightarrow x = y$$

- Induction principles:

$$\begin{aligned} &\forall P : Nat \rightarrow Bool. (P(0) \wedge \forall x : Nat. P(x) \Rightarrow P(s(x))) \\ &\Rightarrow \forall x : Nat. P(x) \end{aligned}$$

Induction and Recursion

Induction is a central proof technique

- For proving properties that are not provable from first-order axioms
- Induction requires an underlying inductive structure, such as inductively defined data structures
- Essential for proving properties of recursive functions

Recursion and Induction are two related concepts:

- Induction principle has
 - a base case
 - an inductive step, based on an induction hypothesis
- Recursive definition of a function consists of
 - a termination case
 - one or more recursive calls

This correspondence is reflected in the proof structure.

Abstract Datatypes

- Abstract datatypes (ADT): important means to model structures, operations and properties
 - built up from constructors
 - operations on the structure
 - properties of structure and operations
- ADTs abstract from concrete realization / implementation of the structure
 - e. g. pointer manipulation for lists or stacks
- particularly important for inductive structures
- Properties of ADTs usually are of algebraic nature, i.e. they can be expressed through algebraic equations.

Example: Binary Trees

Example: abstract datatype of **binary trees** $BTree(T)$ of elements of some type T

- constructors:
 - empty tree: *empty*
 - node containing an element x , and left and right subtrees l and r :
 $node(x, l, r)$
- accessors:
 - element of a node $key(b) : T$
 - left und right subtree: $left(b) : BTree(T)$ $right(b) : BTree(T)$
- Axiomatization

$$\forall x, l, r. \neg(empty = node(x, l, r)) \quad (\text{constr})$$

$$\forall x, y, l, k, r, s. node(x, l, r) = node(y, k, s) \Rightarrow \\ x = y \wedge l = k \wedge r = s \quad (\text{inj})$$

$$\forall x, l, r. key(node(x, l, r)) = x \quad (\text{key})$$

$$\forall x, l, r. left(node(x, l, r)) = l \quad (\text{left})$$

$$\forall x, l, r. right(node(x, l, r)) = r \quad (\text{right})$$

Structural Induction

- Structural induction principles follow the structure of the ADT
- Base case(s) for each constructor without inductive arguments
- inductive step(s) for each inductive constructor
 - induction hypotheses for each inductive argument
 - cf. $P(l) \wedge P(r)$ in case of binary trees

$$\begin{aligned} &\forall P : BTree(T) \rightarrow Bool. \\ &\quad (P(empty) \wedge \\ &\quad \forall x : T, l, r : BTree(T). \\ &\quad \quad P(l) \wedge P(r) \Rightarrow P(node(x, l, r))) \\ &\Rightarrow \forall b : Btree(T). P(b) \end{aligned}$$

- Induction principles can be generated automatically from ADT specification

Recursive Functions

- Recursive functions may be introduced by axioms or by function definitions.
- Axioms can potentially lead to inconsistency
- Definitions of recursive function should be **conservative**, i.e. should not increase the set of provable sentences of a theory.
- Common first-order logic assumes that all functions are total.
- Non-terminating recursive functions introduce partiality:
Options for handling partiality:
 - Treat partiality within the logic, such as
 - LPF – “Logic of Partial Functions” [Barringer, Chen, Jones 1984]
 - LCF – “Logic of Computable Functions”, a many-valued logic with a truth values *undefined* [Scott 1969, Milner 1972].
 - Force all functions to be total by requiring proof of termination for all recursive functions.

Well-Founded Relations

Definition (Well-Founded Relation)

A relation \succ defined on a set S is called **well-founded**, if every non-empty subset M of S has a minimal element m with respect to \succ , i. e., there is no other element x with $m \succ x$.

A **well-founded set** S is a set for which a well-founded relation exists.

An equivalent characterization is that there are no infinitely decreasing chains in S

$$x_1 \succ x_2 \succ \dots x_n \succ \dots$$

Examples of well-founded relations:

- The usual order $>$ on natural numbers
- Sub-structure relations on inductively defined datatypes

Relevance of Well-Founded Relations

Well-founded relations are relevant

- to verify the termination of recursively defined functions
informal arguments for termination that decrease a counter until a termination criteria is reached can be expressed by way of a measure function
- as a foundation for inductive proofs: well-founded induction
the characterization of well-founded relations through minimal elements (i. e., no infinitely decreasing chains) yields another proof principle

PVS: Specifications and Proofs

- PVS specification language
- PVS prover

PVS Specifications

Specifications are structured in PVS into so-called **theories**.

Theories essentially have the following form:

```
<name> [<formal parameters>] : THEORY
BEGIN
  ASSUMING
    <assumptions>
  ENDASSUMING
  IMPORTING <theories ...>
  <declarations, definitions, formulae>
END
```

- A theory combines related declarations, definitions and formulas (modular structure)
- A theory may have types and constants as parameters
- Parameters may be semantically constrained by *assumptions*

Elements of PVS Theories

- Types, both predefined and user-defined, including
 - function types, including higher-order types:
`[nat -> [nat -> nat]]`
 - tuple, array and record types
 - (predicative) sub-types: `posnat : TYPE = {x:nat | x>0}`
 - Dependent types
- Declarations of constants and (typed) variables
- Definitions of functions
- Formulae: expressions of type `bool`, introduced by keywords `AXIOM`, `LEMMA`, `THEOREM` (etc.)
Axioms are assumed to be true, all others must be proved.

Example: PVS theory group

```
group : THEORY
BEGIN
  G : TYPE+          % -- indicates non-empty type
  e : G
  i : [G -> G]
  * : [G,G -> G]      % -- infix operator
  x,y,z : VAR G
  associative : AXIOM
    (x * y) * z = x * (y * z)
  id_left : AXIOM
    e * x = x
  inverse_left : AXIOM
    i(x) * x = e
  inverse_associative : THEOREM
    i(x) * (x * y) = y
END group
```

PVS Specification Language: Expressions

The PVS specification language provides a rich expression language, in effect the equivalent of a full-fledged functional programming language.

- Builds on (simply-typed) λ -calculus
- Boolean connectives and quantifiers for expressions of type `bool`.
- Several forms of conditionals
- Operations on records: field selection and update
- Operations on tuples and arrays: tuple component selection and update
- Function updates, similar to array updates
- Local bindings of variables (`LET-IN` and `WHERE`)
- and more

Predicative Subtypes

```
nat_to_10: TYPE = {x:nat | x <= 10}
posint    : TYPE = {x:integer | x > 0}
pair      : TYPE = {(s:S,t:T) | P(t) => Q(s)}

p : pair = (s0,t0) % -- generates sub-type TCC:
                    %    p_TCC1: OBLIGATION P(t0) => Q(s0)
f : [pair -> X]
f((s0,t0))        % -- generates sub-type TCC, too

Q  : [T -> bool]
Qt : TYPE = (Q)    % -- sub-type of T, containing
                    %    those elements that satisfy Q
```

- following the `|` there can be any (Boolean) formula
- every predicate can be turned into a type using the `()`-notation

Predicative subtypes provide powerful tool for compact specification.

Integration of Specification, Type checking and Proof

Semantic constraints, such as those used in the declaration of predicative subtypes, can often not be statically checked (like common typing) but require proofs.

↪ generation of **type-correctness conditions** (TCCs)

- sub-type TCCs are generated when elements of the sub-type are declared or expected
- termination conditions for recursive function definitions
- ... and various other conditions

TCCs are additional proof obligations that must be discharged to complete a proof.

The PVS system provides support for handling TCCs in the proof management (s. below) and by discharging simple TCCs automatically.

PVS prover

- PVS prover implements the sequent calculus and the other aspects of logic presented in the foundation section.
- PVS prover includes decision procedures for standard decidable theories (linear arithmetic, propositional formulas, ...)
- The PVS prover is primarily interactive: the user guides the proof by issuing prover commands
- Prover commands invoke
 - (combinations of) rules of the sequent calculus
 - instantiations of quantified formulas
 - rules of lambda calculus (e.g. beta reduction), higher-order logic (e.g. extensionality)

PVS prover (2)

- Commands for
 - term rewriting
 - expansion of function calls (replacing function name by definition body)
 - introduction of lemmata into a proof
 - invocation of decision procedures
 - invocation of induction formulas
- Programming of complex proof strategies in a (rudimentary) strategy language

PVS prover (3)

- Sequents are being displayed to the user in readable form.
- A PVS proof is a **proof tree**; construction starts at the root, which is the sequent to be proved
- Proofs are constructed by applying **proof rules**, which can generate one or more subgoals
- Only one subgoal is displayed at a time: the **current sequent**; proof rules apply to that goal
- Subgoals can be worked on in any order
- Various commands for
 - navigating within the (partial) proof tree
 - selecting a sequent as the current one
 - Undoing the proof commands
 - Aborting unfinished proof attempts
- If a branch is proved, the focus moves on to the next open subgoal
- If there is no more, then the initial formula is proved
- Proofs can be re-run, e. g. after modification of the theory

Proof Management

- PVS stores proof scripts in a separate file
- possible to store multiple versions of proofs
- PVS keeps track of **proof status** of a formula:
 - **untried**: no proof stored
 - **unfinished**: proof has been aborted
 - **proved – incomplete**: formula is proved, but depends on unproved formulae
 - **proved – complete**: formula is proved, as well as all formulae it depends on
 - **unchecked**: formula has been proved, but theory has been modified afterwards
- Display proof status of a formula
- To store a theory, or a hierarchy of theories, together with all proofs in a single **dump**-file

Verification with PVS

Using PVS to verify properties of a system, algorithms, ... involves the following steps:

- *Modelling and specification:*
 - Develop a set of PVS theories that capture the essence of the behaviour of the system, the algorithms, ... at an appropriate abstract level
 - PVS provides a rich collection of complete PVS theories (mainly for mathematical structures) with proved theorems
 - Hence users need not start from “first principles”.
- Specify the desired/expected properties as theorems.
- Develop proofs of the theorems.
 - PVS provides support for keeping track of proved and unproved theorems, lemmas, TCCs ...
- Completed proofs can be “replayed” and adapted to changed specifications.

Applications of PVS

The PVS system has been used to specify and verify properties in a large variety of application areas. These include:

- formalization of mathematical concepts and proofs
 - for analysis, graph theory, number theory, etc.
- embedding of formalisms, such as
 - I/O automata
 - various forms of modal and temporal logics
- verification of
 - hardware
 - sequential and distributed algorithms, fault-tolerant algorithms
 - real-time and hybrid systems
 - safety and security of applications
 - compiler correctness
- use as a back-end verification tool for computer algebra and code verification systems

For virtually any application area of formal methods there exists applications using PVS.

References

- Treatment of partiality in logics
 - Howard Barringer, J. H. Cheng, Cliff B. Jones: A Logic Covering Undefinedness in Program Proofs. Acta Inf. 21: 251-269 (1984)
 - Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. Theoretical Computer Science, 121:411–440, 1993. Annotated version of the 1969 manuscript.
 - Robin Milner. Logic for computable functions; description of a machine implementation. Technical Report STAN-CS-72-288, A.I. Memo 169, Stanford University, 1972.
- PVS documentation
<http://pvs.csl.sri.com/documentation.shtml>
- PVS examples
 - collection of examples at SRI:
<ftp://ftp.csl.sri.com/pub/pvs/examples/>
 - PVS developments at NASA LaRC, incl. libraries:
<http://shemesh.larc.nasa.gov/fm/fm-pvs.html>
 - application of PVS in various areas, incl. fault tolerance:
<http://www.uni-ulm.de/in/ki/PVS/>

Part III

Static Program Analysis

Cinzia Bernardeschi and Paolo Masci

Part of these slides is from the course “Abstract interpretation and static analysis” held at the International Winter School on Semantics and Applications, Uruguay, 2003, by David Schmidt

Static program analysis

Static program analysis is a set of techniques to compute reliable approximate information about the dynamic behaviour of programs

- program properties are determined without actually executing the program

Program analysis should be semantics based

- the information obtained from the analysis should be proved to be correct wrt a semantics of the programming language

Motivation

The concrete semantics of programs formalises the set of all possible executions in all possible environments

- non trivial questions about the concrete semantics of a program are undecidable

Static analysis considers an abstract semantics, that is a superset of the concrete program semantics

- the concrete domain of values and operations are replaced by an abstract domain and corresponding abstract operations

Static program analysis is a sound, finite and approximate calculation of the execution semantics of a program

Applications

Main applications:

- optimisation of programs at compile-time
- formal verification of program properties

Examples:

- live variables
- reaching definitions
- code motion
- constant propagation
-
- type correctness of programs
- abstract testing and safety checking
- assertion checking and discovery
- ...

Data-flow analysis

Many automated static program analyses are based on data-flow analysis

Data-flow analysis: set of techniques suitable to derive information about the flow of data along program execution paths

A data-flow analysis typically requires the following steps:

- build program representation
- map program properties to abstract values organised into an algebraic structure (semilattice, lattice)
- associate a data-flow state (of abstract values) to each program point
- perform forward/backward data-flow analysis

Data-flow framework

A data-flow analysis framework $(D, \mathcal{V}, \sqcap, F)$ consists of:

- A direction D of the data-flow, which is either *forwards* or *backwards*
- A semilattice (\mathcal{V}, \sqcap) , where \mathcal{V} is a domain of value and \sqcap is the meet operator
 - the meet operator is idempotent, commutative, associative
 - \mathcal{V} has a top element \top : for all x , $\top \sqcap x = x$
 - optionally, \mathcal{V} may have a bottom element \perp : for all x , $\perp \sqcap x = \perp$
- A family F of transfer functions $f : \mathcal{V} \rightarrow \mathcal{V}$;
 - F has an identity function $I(x) = x$ for all x
 - F is closed under composition: given two functions f and g in F , $g(f(x))$ is in F

Data-flow framework (contd.)

The pair (\mathcal{V}, \leq) is a poset, where \leq is a partial order relation defined as $x \leq y$ iff $x \sqcap y = x$

- it is also convenient to have a $<$ relation: $x < y$ iff $x \leq y$ and $x \neq y$

Given a poset (\mathcal{V}, \leq) , an ascending chain is a sequence

$$x_1 < x_2 < \cdots < x_n$$

The height of a semilattice \mathcal{V} is the largest number of $<$ in any ascending chain

$(D, \mathcal{V}, \sqcap, F)$ is monotone if

for all x and y in \mathcal{V} and f in F , $x \leq y$ implies $f(x) \leq f(y)$

or, equivalently,

for all x and y in \mathcal{V} and f in F , $f(x \sqcap y) \leq (f(x) \sqcap f(y))$

$(D, \mathcal{V}, \sqcap, F)$ is distributive if

for all x and y in \mathcal{V} and f in F , $f(x \sqcap y) = (f(x) \sqcap f(y))$

Program representation

In data-flow analyses, programs are usually represented with a control flow graph.

Control flow graph: a directed graph containing the control dependences among instructions of a program

- each node s_i in the control flow graph represents an instruction
- there is an edge $s_i \rightarrow s_j$ between two instructions if s_j can be executed immediately after s_i
- special *ENTRY* and *EXIT* nodes

Property: the control flow graph is a conservative approximation of the control flow of a program.

Data-flow values and transfer function

Data-flow values are associated with each instruction in the program

- an input state $IN[s_i]$ is the data-flow value *before* statement s_i
- an output state $OUT[s_i]$ is the data-flow value *after* statement s_i

A transfer function f_{s_i} is associated with statement s_i

- each execution of a code statement transform an input state into an new output state

Objective of the analysis:

- find a solution to the set of constraints on $IN[s_i]$ and $OUT[s_i]$, for all s_i

Constraints

Constraints of the analysis

- transfer functions
- control flow graph of the program

Transfer function constraints:

- *forward-flow* problem: $OUT[s_i] = f(IN[s_i])$
- *backward-flow* problem: $IN[s_i] = f(OUT[s_i])$

Constraints (contd.)

Control flow constraints:

- *forward-flow* problem:
 $IN[s_i] = \sqcap_{s_j} OUT[s_j]$, where s_j is a predecessor of s_i
- *backward-flow* problem:
 $OUT[s_i] = \sqcap_{s_j} IN[s_j]$, where s_j is a successor of s_i

Note that \sqcap depends on the type of analysis.

- in the forwards analysis, if we have information about the set of constants that may be assigned to a variable, \sqcap is the union
- in the backwards analysis, if we have information about the live variable analysis, \sqcap is the union

Equations are solved with least fixpoint iteration

Iterative algorithm

Input:

- a data-flow graph with special *ENTRY* and *EXIT* nodes
- a data-flow framework $(D, \mathcal{V}, \sqcap, F)$
- v_{ENTRY} and v_{EXIT} in \mathcal{V} , which represent the boundary conditions for forward and backward analyses

Output:

- values in \mathcal{V} for $IN[s_i]$ and $OUT[s_i]$ for each s_i

Iterative algorithm (contd.)

Iterative algorithm for forward analysis:

- ① $OUT[ENTRY] = v_{ENTRY};$
- ② **for** (each instruction s_i other than $ENTRY$) $OUT[s_i] = \top;$
- ③ **while** (changes to any OUT occur)
- ④ **for** (each instruction s_i other than $ENTRY$) {
- ⑤ $IN[s_i] = \sqcap_{s_j} OUT[s_j]$ for all s_j predecessors of s_i ;
- ⑥ $OUT[s_i] = f_{s_i}(IN[s_i]);$
- }

Iterative algorithm (contd.)

Iterative algorithm for backward analysis:

- ① $IN[EXIT] = v_{EXIT};$
- ② **for** (each instruction s_i other than $EXIT$) $IN[s_i] = \top$
- ③ **while** (changes to any IN occur)
- ④ **for** (each instruction s_i other than $EXIT$) {
- ⑤ $OUT[s_i] = \sqcap_{s_j} IN[s_j]$ for all s_j successors of s_i
- ⑥ $IN[s_i] = f_{s_i}(OUT[s_i])$
- }

Iterative algorithm (contd.)

Property 1: if the algorithm converges, the result is a solution to the data-flow equations

Property 2: if the framework is monotone, the solution is the maximum fixpoint of the data-flow equations

Property 3: if the semilattice is monotone and of finite height then the algorithm converges

$IN[s_i]$ represents an abstraction of the set of all possible program states that can be observed before s_i

$OUT[s_i]$ represents an abstraction of the set of all possible program states that can be observed after s_i

Meet-over-path solution

In the data-flow abstraction every path P in the flow graph can be taken.

In the forward framework, we have that:

- $P = ENTRY \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_{k-1} \rightarrow s_k$ (Note that the same instructions may appear several times because of loops.)

Let f_P be the composition of $f_{s_1}, f_{s_2}, \dots, f_{s_{k-1}}$, and $f_P(v_{ENTRY})$ is the data-flow value created by executing path P

- the meet-over-path solution is $MOP[s_i] = \sqcap_P f_P(v_{ENTRY})$, where P is a path from $ENTRY$ to s_i

Meet-over-path solution (contd.)

Paths considered in the *MOP* solution are a superset of all the paths that are possibly executed.

If we take the meet, then $MOP[s_i] \leq IDEAL[s_i]$, where $IDEAL[s_i]$ is the ideal solution.

The solution of the iterative algorithm is safe:

- If the data-flow framework is distributive, the solution given by the iterative algorithm and the *MOP* solution are the same.
- If the data-flow framework is monotone, but not distributive, the solution given by the iterative algorithm is \leq *MOP* solution.

Note that, in symmetry with the meet operator \sqcap (greatest-lower-bound) on posets, we may define a join operator \sqcup (least-upper-bound) on posets.

Example: the Java bytecode verifier

The Java bytecode verifier performs a static analysis of the Java bytecode for type correctness, stack overflows and underflows, ...

The verification is performed method-per-method: when verifying a method, the other methods are assumed to be correct

A forward data-flow analysis is used

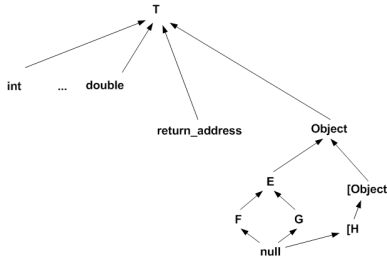
Values are abstracted into types

The execution of instructions is performed on types instead of real values

Data-flow analysis

The poset $(\mathcal{V}, \sqsubseteq)$ is defined

- \mathcal{V} is the set of Java types plus a type *null*, which represents the type of *null* references, and \top , which represents the content of uninitialised registers (i.e., any value)
- \sqsubseteq is the subtyping ordering relation given by the “assignable to” relation of Java



Data-flow analysis (contd.)

The transfer function is defined as

$$f_{s_i} : \langle S, R \rangle \rightarrow \langle S', R' \rangle$$

where $\langle S, R \rangle$ is the stack type and register type before executing instruction s_i , and $\langle S', R' \rangle$ is the stack type and register type after executing instruction s_i .

Examples of transfer functions:

$$\begin{aligned} \text{iload_j: } & \langle S, R \rangle \rightarrow \langle \text{int} \cdot S, R \rangle \\ & \text{if } 0 \leq j < M_{\text{reg}}, R(j) = \text{int and } \#S < M_{\text{stack}} \end{aligned}$$

$$\begin{aligned} \text{istore_j: } & \langle \text{int} \cdot S, R \rangle \rightarrow \langle S, R[j/\text{int}] \rangle \\ & \text{if } 0 \leq j < M_{\text{reg}} \end{aligned}$$

$$\begin{aligned} \text{getfield C.f: } & \langle C' \cdot S, R \rangle \rightarrow \langle t \cdot S, R \rangle \\ & \text{if C.f has type } t \text{ and } C \sqsubseteq C' \end{aligned}$$

Data-flow equations

The state before instruction s_i is taken to be the least-upper-bound on the states after all predecessors of s_i . The least-upper-bound represents the smallest common supertype.

$IN[s_i] = \sqcup_{s_j} OUT[s_j]$, where s_j is a predecessor of s_i

Equations are solved by standard fixpoint iteration using a worklist:

- an instruction s_i is taken from the worklist
- the after state $OUT[s_i] = f_{s_i}(IN[s_i])$ is computed
- $IN[s_j] = \sqcup_{s_i} (IN[s_i], OUT[s_i])$ for all s_j successor of s_i
- s_j is inserted in the worklist if $IN[s_j]$ changed

Fixpoint reached when worklist is empty; in this case, the verification succeeds.

Verification fails if *i*) a state with no transition is encountered, *ii*) one of the least-upper-bounds is undefined.

Static analysis and abstract interpretation

- Abstract interpretation is a method for designing approximate semantics of programs
- The concept of abstract interpretation was introduced by Patrick and Radhia Cousot in order to formalise static program analysis, in “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints” (POPL77)
- The abstract interpretation approach is based on the use of Galois connections to establish a correspondence between the domain of concrete properties and the domain of abstract properties
- Static analysers can be constructively derived from the abstract semantics

Foundations of abstract interpretation

Abstract interpretation is based on lattices, continuous functions and Galois connections.

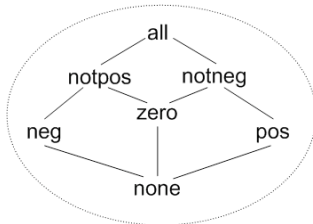
A complete lattice $\mathcal{L} = \langle D, \sqsubseteq, \top, \perp, \sqcup, \sqcap, \rangle$ consists of:

- a set D and a partial ordering \sqsubseteq on D
- a smallest element \perp , such that $\perp \sqsubseteq d$ for all $d \in D$, and a greatest element \top , such that $d \sqsubseteq \top$ for all $d \in D$
- a least-upper-bound operation \sqcup such that, for all $S \subseteq D$, $d \sqsubseteq \sqcup S$ for all $d \in S$, and, for all other upper bounds $c \in D$ such that $d \sqsubseteq c$ for all $d \in S$, we have that $\sqcup S \sqsubseteq c$
- a greatest lower bound operation \sqcap , defined dually to the above: $\sqcap S \sqsubseteq d$ for all $d \in S$, and, when $c \sqsubseteq d$ for all $d \in S$, we have that $c \sqsubseteq \sqcap S$

Example

An example of complete lattice, with

$D = \{all, notpos, notneg, zero, neg, pos, none\}$:



$\sqcap \{notpos, notneg\} = zero$

$\sqcup \{zero, notpos, notneg\} = all$

$\sqcap \{\} = all$

$\sqcup \{\} = none$

Monotonic and chain continuous functions

Conventional computation employs monotonic and ω -continuous functions

Monotonic function: given two complete lattices \mathcal{A} and \mathcal{B} , a function $f : \mathcal{A} \rightarrow \mathcal{B}$ is monotonic iff for all $a, a' \in \mathcal{A}$, $a \sqsubseteq_{\mathcal{A}} a'$ implies $f(a) \sqsubseteq_{\mathcal{B}} f(a')$

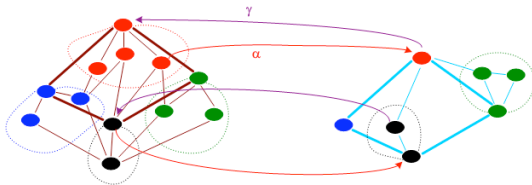
ω -continuous function: given two complete lattices \mathcal{A} and \mathcal{B} and a ω -chain $a_0 \sqsubseteq_{\mathcal{A}} a_1 \sqsubseteq_{\mathcal{A}} \cdots \sqsubseteq_{\mathcal{A}} a_i \sqsubseteq_{\mathcal{A}} a_{i+1} \sqsubseteq_{\mathcal{A}} \cdots$, a function $f : \mathcal{A} \rightarrow \mathcal{B}$ is ω -continuous iff $\bigsqcup_{i \geq 0} f(a_i) = f(\bigsqcup_{i \geq 0} a_i)$.

- A monotonic function preserves the precision information of its argument
- A ω -continuous function preserves the limit of information in a chain

Galois connections

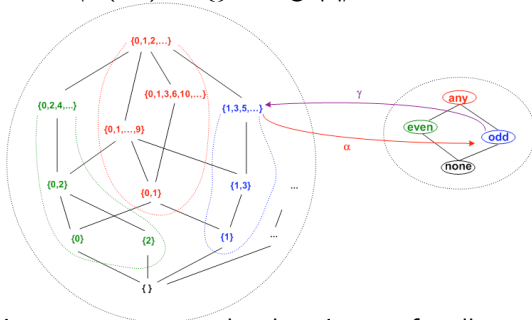
Given a complete lattice of *concrete* execution data \mathcal{C} , and a simpler complete lattice of *abstract* data \mathcal{A} , the function $\alpha : \mathcal{C} \rightarrow \mathcal{A}$ acts like a homomorphism when we study the operations on \mathcal{C} . Let $\gamma : \mathcal{A} \rightarrow \mathcal{C}$ be the inverse of α .

Galois connection: for complete lattices, \mathcal{C} and \mathcal{A} , and monotonic functions, $\alpha : \mathcal{C} \rightarrow \mathcal{A}$ and $\gamma : \mathcal{A} \rightarrow \mathcal{C}$, the pair $\langle \alpha, \gamma \rangle$ forms a Galois connection, written $\mathcal{C} \langle \alpha, \gamma \rangle \mathcal{A}$, iff $c \sqsubseteq_{\mathcal{C}} \gamma \circ \alpha(c)$ and $\alpha \circ \gamma(a) \sqsubseteq_{\mathcal{A}} a$.



Galois connections (contd.)

A complete lattice $\langle \mathcal{P}(\text{Int}), \subseteq, \{\}, \text{Int}, \cup, \cap \rangle$, and an abstraction of it:



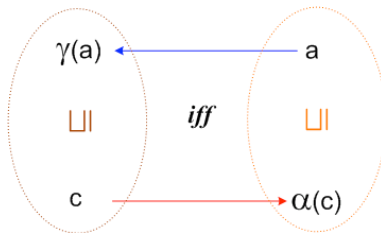
α and γ are inverse maps on each others image: for all $c \in \gamma[\mathcal{A}]$, $c = \gamma \circ \alpha(c)$, and that for all $a \in \alpha[\mathcal{C}]$, $a = \alpha \circ \gamma(a)$

- α is ω -continuous and even preserves \sqcup for arbitrary sets in \mathcal{C}
- γ preserves \sqcap for arbitrary sets in \mathcal{A}
- each map uniquely defines the other:

$$\gamma(a) = \sqcup \{c \mid \alpha(c) \sqsubseteq_{\mathcal{A}} a\} \text{ and } \alpha(c) = \sqcap \{a \mid c \sqsubseteq_{\mathcal{C}} \gamma(a)\}$$

Alternative characterisation of Galois connections

For complete lattice \mathcal{C} and \mathcal{A} , the pair $\langle \alpha : \mathcal{C} \rightarrow \mathcal{A}, \gamma : \mathcal{A} \rightarrow \mathcal{C} \rangle$ is a Galois connection when, for all $c \in \mathcal{C}$ and $a \in \mathcal{A}$, $c \sqsubseteq_{\mathcal{C}} \gamma(a)$ iff $\alpha(c) \sqsubseteq_{\mathcal{A}} a$.



We can prove that

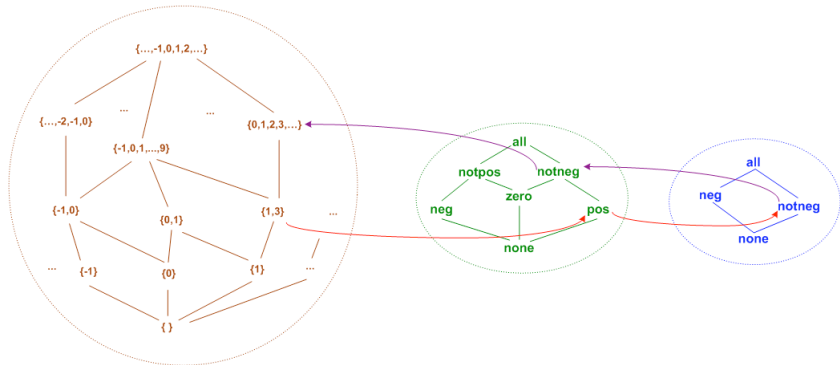
- both α and γ are monotonic
- $c \sqsubseteq_{\mathcal{C}} \gamma \circ \alpha(c)$
- $\alpha \circ \gamma(a) \sqsubseteq_{\mathcal{A}} a$.

Properties of Galois connections

Galois connections are closed under composition, product, and so on:

- If $\mathcal{C} \langle \alpha, \gamma \rangle \mathcal{D}$ and $\mathcal{D} \langle \alpha', \gamma' \rangle \mathcal{E}$ are Galois connection, then so is $\mathcal{C} \langle \alpha' \circ \alpha, \gamma' \circ \gamma \rangle \mathcal{E}$
- If $\mathcal{C}_i \langle \alpha_i, \gamma_i \rangle \mathcal{D}_i$ is a Galois connection for all $i \in I$, then so is $\prod_{i \in I} \mathcal{C}_i \langle \prod_{i \in I} \alpha_i, \prod_{i \in I} \gamma_i \rangle \prod_{i \in I} \mathcal{D}_i$
- If $\mathcal{C} \langle \alpha_{\mathcal{C}}, \gamma_{\mathcal{C}} \rangle \mathcal{C}'$ and $\mathcal{D} \langle \alpha_{\mathcal{D}}, \gamma_{\mathcal{D}} \rangle \mathcal{D}'$ are Galois connection, then so is $\mathcal{C} \rightarrow \mathcal{D} \langle (\lambda f. \alpha_{\mathcal{D}} \circ f \circ \gamma_{\mathcal{C}}), (\lambda f'. \gamma_{\mathcal{D}} \circ f' \circ \alpha_{\mathcal{C}}) \rangle \mathcal{C}' \rightarrow \mathcal{D}'$.

Composition of Galois connections



Utility of Galois connections

Given the definition of $\gamma : \mathcal{A} \rightarrow \mathcal{C}$, we can mechanically synthesise its adjoint, $\alpha(c) = \sqcap \{a \mid c \sqsubseteq_{\mathcal{C}} \gamma(a)\}$

Dually, given α , we can synthesise γ as $\gamma(a) = \sqcup \{c \mid \alpha(c) \sqsubseteq_{\mathcal{A}} a\}$

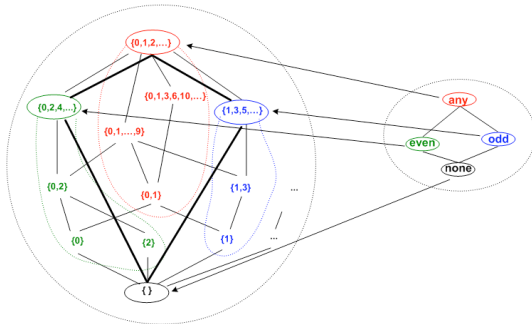
Many mathematical properties about α can be expressed in terms of its adjoint γ (and vice versa)

Since we use $\alpha : \mathcal{C} \rightarrow \mathcal{A}$ as a homomorphism from \mathcal{C} to \mathcal{A} , abstract operations can be synthesised with α and its adjoint γ :

- for each $f : \mathcal{C} \rightarrow \mathcal{C}$ we can synthesise $f^{\#} : \mathcal{A} \rightarrow \mathcal{A}$ such that α is a homomorphism with respect to f and $f^{\#}$.

Closure maps

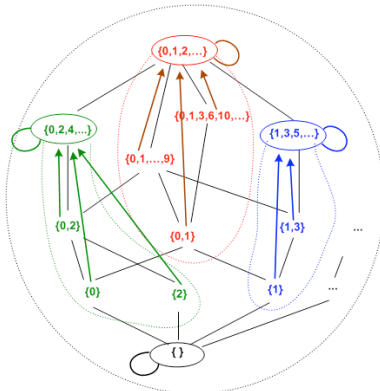
For $\mathcal{C}\langle\alpha, \gamma\rangle\mathcal{A}$, it is common that \mathcal{A} embeds into \mathcal{C} as a sublattice. Elements in \mathcal{A} are tokens that name distinguished sets in \mathcal{C} .



Closure map: $\rho : \mathcal{C} \rightarrow \mathcal{C}$ is a closure map if it is *i)* monotonic;
ii) extensive: $c \sqsubseteq_{\mathcal{C}} \rho(c)$ for all $c \in \mathcal{C}$; *iii)* idempotent: $\rho \circ \rho = \rho$.

Closure maps (contd.)

- every Galois connection $\mathcal{C}\langle\alpha, \gamma\rangle\mathcal{A}$, defines a closure map $\gamma \circ \alpha$.
- every closure map $\rho : C \rightarrow \mathcal{C}$, defines the Galois connection $\mathcal{C}\langle\rho, id\rangle\rho[C]$



Moore families

Given \mathcal{C} , we can define a closure map on \mathcal{C} by choosing a subset of elements of \mathcal{C} closed under greatest-lower-bounds

Moore family: $\mathcal{M} \subseteq \mathcal{C}$ is a Moore family iff, for all $S \subseteq \mathcal{M}$, $(\sqcap S) \in \mathcal{M}$

We can define a closure map as $\rho(c) = \sqcap \{c' \in \mathcal{M} \mid c \sqsubseteq_{\mathcal{C}} c'\}$

For a closure map $\rho : \mathcal{C} \rightarrow \mathcal{C}$, its image $\rho[\mathcal{C}]$ is a Moore family

Hence, given \mathcal{C} , we can define an abstract interpretation by selecting some $\mathcal{M} \subseteq \mathcal{C}$ that is a Moore family

Closed binary relations

Often a Galois connection uses a powerset for its concrete domain, that is, $\mathcal{P}(D) \langle \alpha, \gamma \rangle \mathcal{A}$

Given an unordered set \mathcal{D} and complete lattice \mathcal{A} , it is natural to relate the elements in D to those in \mathcal{A} by a binary relation, $\mathcal{R} \subseteq D \times \mathcal{A}$, such that $(d, a) \in \mathcal{R}$ means “d has property a”

We write this as $d\mathcal{R}a$

Example

$D = \text{Int}$, and $\mathcal{A} = \{\text{none}, \text{neg}, \text{pos}, \text{zero}, \text{nonneg}, \text{nonpos}, \text{any}\}$.

Then $2\mathcal{R}\text{nonneg}$, $2\mathcal{R}\text{pos}$ and $2\mathcal{R}\text{any}$.

We can define $\gamma : \mathcal{A} \rightarrow \mathcal{P}(D)$ as $\gamma(a) = \{d \in D \mid d\mathcal{R}a\}$,
for example, $\gamma(\text{nonneg}) = \{0, 1, 2, \dots\}$.

Utility of \mathcal{R}

We can check if γ is the upper adjoint of a Galois connection by showing that $\gamma[\mathcal{A}]$ defines a Moore family. This can be done directly upon \mathcal{R} :

Proposition: $\mathcal{R} \subseteq D \times \mathcal{A}$ defines a Galois connection between $\mathcal{P}(D)$ and \mathcal{A} iff i) \mathcal{R} is U-closed: $c\mathcal{R}a$ and $a \sqsubseteq_{\mathcal{A}} a'$ imply $c\mathcal{R}a'$; ii) \mathcal{R} is G-closed: $c\mathcal{R} \sqcap \{ a \mid c\mathcal{R}a \}$.

If \mathcal{R} defines a Galois connection, then we have:

- for all $a \in \mathcal{A}$ and $\mathcal{C} \in \mathcal{P}(D)$,
 $\mathcal{C} \subseteq \gamma(a)$ iff $\alpha(\mathcal{C}) \sqsubseteq_{\mathcal{A}} a$ iff $(c\mathcal{R}a, \text{ for all } c \in \mathcal{C})$.

This is the definition of a Galois connection and, in this sense, \mathcal{R} is a Galois connection.

Abstract-domain building

Given an unordered set D of concrete data values and a set of properties \mathcal{A} , we can relate properties $a \in \mathcal{A}$ to elements $d \in D$ via a UG-closed binary relation $\mathcal{R}_D \subseteq D \times \mathcal{A}$

- 1 Define $\gamma : \mathcal{A} \rightarrow \mathcal{P}(D)$ as $\gamma(a) = \{d \mid d\mathcal{R}_D a\}$.
- 2 Define this partial ordering on \mathcal{A} : $a \sqsubseteq a'$ iff $\gamma(a) \subseteq \gamma(a')$. If there are distinct $a, a' \in \mathcal{A}$ such that $\gamma(a) = \gamma(a')$, then merge them. This forces U-closure.
- 3 Ensure that $\gamma[\mathcal{A}]$ is a Moore family by adding greatest-lower-bound elements to \mathcal{A} as needed. This forces G-closure.
- 4 Use the existing machinery to define the Galois connection between $\mathcal{P}(D)$ and \mathcal{A} .

Sound approximation

For Galois connection $\mathcal{C} \langle \alpha, \gamma \rangle \mathcal{A}$, and functions $f : \mathcal{C} \rightarrow \mathcal{C}$, we can say that $f^\# : \mathcal{A} \rightarrow \mathcal{A}$ is a *sound approximation* of f iff

$$\begin{aligned} (\alpha \circ f)(c) &\sqsubseteq_{\mathcal{A}} (f^\# \circ \alpha)(c), \text{ for all } c \in \mathcal{C} \\ \text{iff} \\ (f \circ \gamma)(a) &\sqsubseteq_{\mathcal{C}} (\gamma \circ f^\#)(a), \text{ for all } a \in \mathcal{A} \end{aligned}$$

Note that α acts like a “semi-homomorphism” with respect to f and $f^\#$:

$$\begin{array}{ccc} c & \xrightarrow{\alpha} & \alpha(c) \\ f \downarrow & & \downarrow f^\# \\ f(c) & \xrightarrow{\alpha} & \alpha(f(c)) \sqsubseteq f^\#(\alpha(c)) \end{array}$$

Synthesising f^\sharp from f

Given the Galois connection $\mathcal{C}\langle\alpha, \gamma\rangle\mathcal{A}$, and operation $f : \mathcal{C} \rightarrow \mathcal{C}$, the most precise $f_{best}^\sharp : \mathcal{A} \rightarrow \mathcal{A}$ that is sound with respect to f is

$$f_{best}^\sharp = \alpha \circ f \circ \gamma$$

Proposition: f^\sharp is sound with respect to f iff $f_{best}^\sharp \sqsubseteq_{\mathcal{A} \rightarrow \mathcal{A}} f^\sharp$.

Note that

- $f \sqsubseteq_{\mathcal{A} \rightarrow \mathcal{A}} g$ iff for all $a \in \mathcal{A}$, $f(a) \sqsubseteq_{\mathcal{A}} g(a)$.
- f_{best}^\sharp has a mathematical definition, and not an algorithmic one; hence, f_{best}^\sharp might not be finitely computable.

Completeness

Given $\mathcal{C}\langle\alpha, \gamma\rangle\mathcal{A}$, we state soundness of $f^\#$ with respect to f as
 $\alpha \circ f \sqsubseteq_{\mathcal{A} \rightarrow \mathcal{A}} f^\# \circ \alpha$ iff $f \circ \gamma \sqsubseteq_{\mathcal{C} \rightarrow \mathcal{C}} \gamma \circ f^\#$.

Definitions:

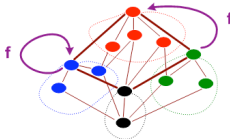
- $f^\#$ is *forwards* (γ) *complete* with respect to f iff $f \circ \gamma =_{\mathcal{C} \rightarrow \mathcal{C}} \gamma \circ f^\#$
- $f^\#$ is *backwards* (α) *complete* with respect to f iff $\alpha \circ f =_{\mathcal{A} \rightarrow \mathcal{A}} f^\# \circ \alpha$

Note that the two completeness notions are not equivalent.

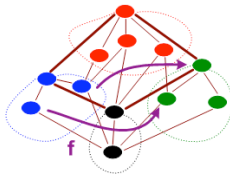
For an $f^\#$ to be (forwards or backwards) complete, it must equal $f_{best}^\# = \alpha \circ f \circ \gamma$. Indeed, the structure of the Galois connection and $f : \mathcal{C} \rightarrow \mathcal{C}$ determines whether $f_{best}^\#$ is complete.

Completeness (contd.)

Forwards (γ) completeness: $f_{best}^\#$ is forwards-complete iff f maps image points of γ to image points of γ , i.e., $f(\gamma[A]) \subseteq \gamma[A]$.



Backwards (α) completeness: $f_{best}^\#$ is backwards-complete iff f maps all points in the same α -equivalence class to points in the same α -equivalence class, i.e., $\alpha(c) = \alpha(c') \rightarrow \alpha(f(c)) = \alpha(f(c'))$.



References

C. Hankin, F. Nielson, H. R. Nielson: “Principles of Program Analysis”, Springer, 1999

A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman: “Compilers: Principles, Techniques, and Tools”, Addison-Wesley, 2006

P. Cousot, R. Cousot: “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints” in POPL77, pages 238–252, Los Angeles, California, 1977

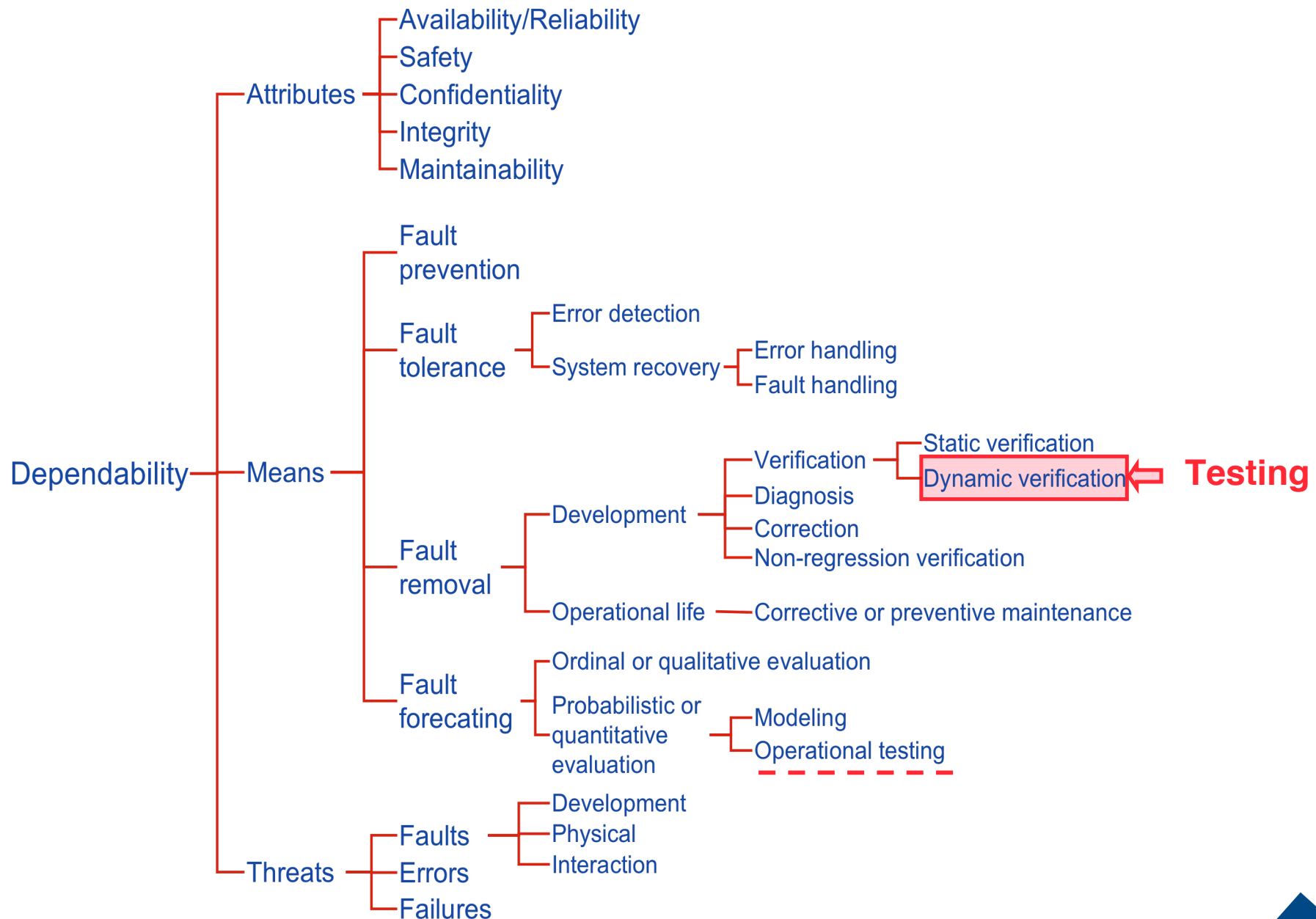
P. Cousot, R. Cousot: “Systematic Design of Program Analysis Frameworks”, in POPL79, pages 269–282, San Antonio, Texas, 1979.

D. Schmidt: Course on *Abstract interpretation and static analysis*, held at the International Winter School on Semantics and Applications, Uruguay, 2003 (<http://santos.cis.ksu.edu/schmidt/Escuela03/>).

Part IV

Introduction to Software Testing

Hélène WAESELYNCK



Software: first failure cause of computing systems

Size: from some (tens) of thousands of code lines to some millions of code lines

Development effort:

0,1-0,5 person.year / KLOC (large software)

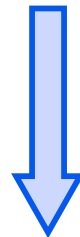
5-10 person.year / KLOC (critical software)

Share of the effort devoted to fault removal:

45-75%

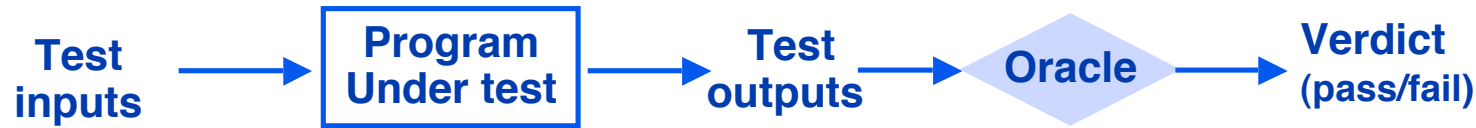
Fault density:

10-200 faults / KLOC created during development



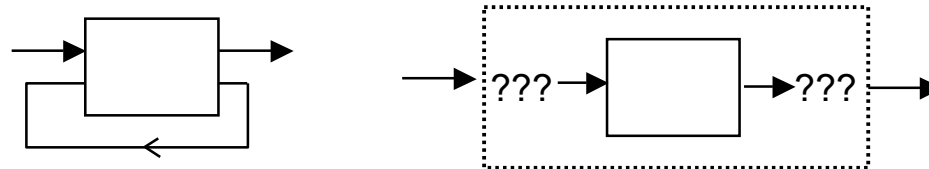
- static analysis
- proof
- model-checking
- *testing*

0,01-10 faults / KLOC residual in operation



○ Issues of **controllability** and **observability**

➤ Examples :



○ **Oracle** problem = how to decide about the correctness of the observed outputs?

➤ Manual computation of expected results, executable specification, back-to-back testing of different versions, output plausibility checks, ...

○ To **reveal a fault**, the following chain of conditions must be met:

- At least one test input activates the fault and creates an error
- The error is propagated until an observable output is affected
- The erroneous output violates an oracle check

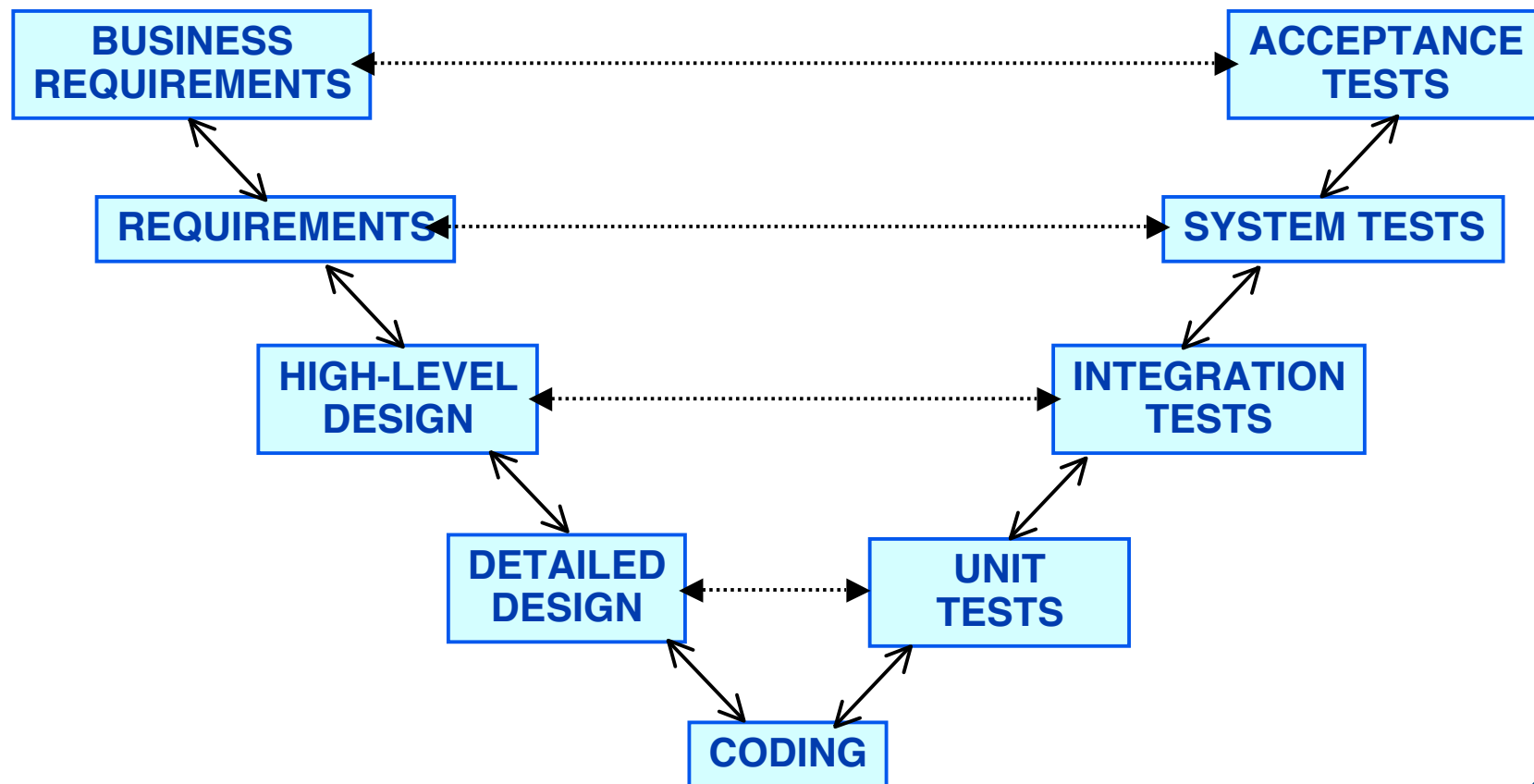
If $x > 0$: output $(x+1)/3 + 1$
Else: output 0

```
Example_function (int x)
BEGIN
  int y, z ;
  IF x ≤ 0 THEN
    z = 0
  ELSE
    y = x-1 ; /* y = x+1 */
    z = (y/3) +1 ;
  ENDIF
  Print(z) ;
END
```

- Activation of the fault if $x > 0$
- Error propagation: incorrect output if $(x \bmod 3) \neq 1$
- Violation of an oracle check:
 - ➔ Expected result correctly determined by the operator \Rightarrow fault revealed
 - ➔ Back-to-back testing of 2 versions, V2 does not contain this fault \Rightarrow fault revealed
 - ➔ Plausibility check $0 < 3z-x < 6 \Rightarrow$ fault revealed if $(x \bmod 3) = 0$

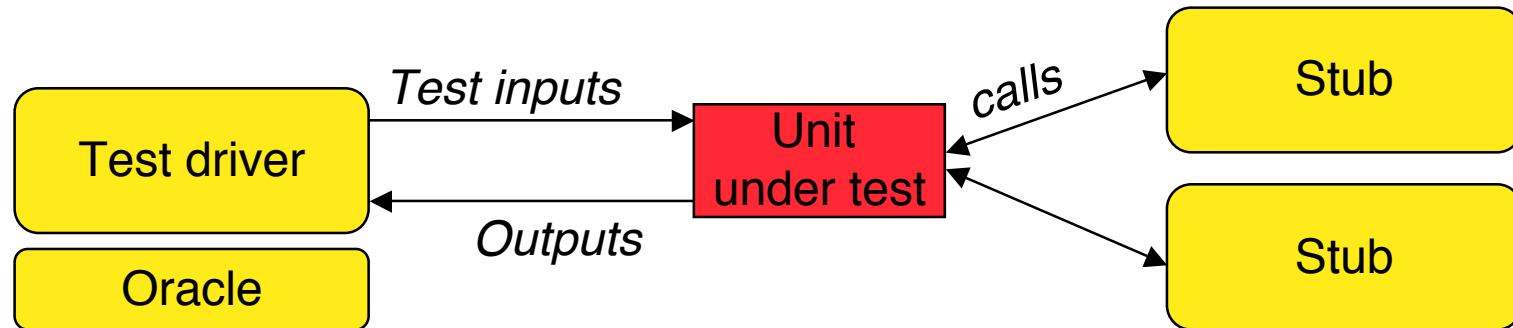
Explicit consideration of testing in the software life cycle

Example: V life cycle



- **Whatever the adopted life-cycle model, it defines a testing process, in interaction with the software development process**
 - Planning test phases associated with development phases
 - Progressive integration strategy (e.g., top-down design, bottom-up testing)
 - Tester/developer independence rules (according to software phase and criticality)
 - Rules guiding choice of test methods to employ (according to software phase and criticality)
 - Procedures for coordinating processes
- **Documents are produced at each testing step**
 - Employed test methods
 - Test sets + oracle
 - Test platform: host machine, target machine emulator, target machine, external environment simulator
 - Other tools: compiler, test tools, drivers and stubs specifically developed
 - Test reports

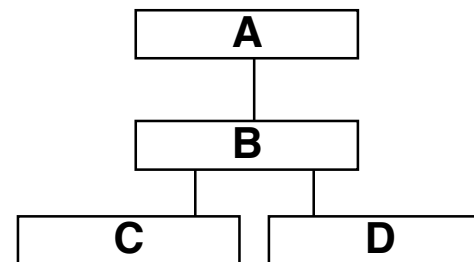
- **Unit testing = testing of an isolated component**



- **Integration testing = gradual aggregation of components**

E.g.: bottom-up strategy

Test {C}, test {D}
 Test {B, C, D}
 Test {A, B, C, D}



😊 no stub to develop

☹️ **High-level components are tested late, while it may be important to test them early**

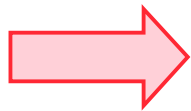
- because they are major components of the system (e.g., GUI)
- to reveal high-level faults (e.g., inadequate functional decomposition)

⇒ **Other strategies : top-down, sandwich**

Test design methods: problem

Exhaustive testing is impractical!

- **Very large, or even infinite, input domain**
 - Testing a simple program processing three 32 bits integers:
 2^{96} possible inputs
 - Testing a compiler: infinite input domain
- **Relevance of the very notion of exhaustiveness?**
 - Elusive faults (Heisenbugs): activation conditions depend on complex combinations of internal state x external requests



**Partial verification using a (small) sample
of the input domain**

Adequate selection?



No model of all possible software faults

Classification of test methods

② \ ①	STRUCTURAL MODEL	FUNCTIONAL MODEL
SELECTIVE CHOICE	deterministic structural	deterministic functional
RANDOM CHOICE	statistical structural	statistical functional

① criterion

The model synthesizes information about the program to be tested.

The criterion indicates how to exploit the information for selecting test data: it defines a set of model elements to be exercised during testing.

② input generation process

Deterministic: selective choice of inputs to satisfy (cover) the criterion.

Probabilistic: random generation according to a probabilistic distribution over the input domain; the distribution and number of test data are determined by the criterion.

Purposes of testing

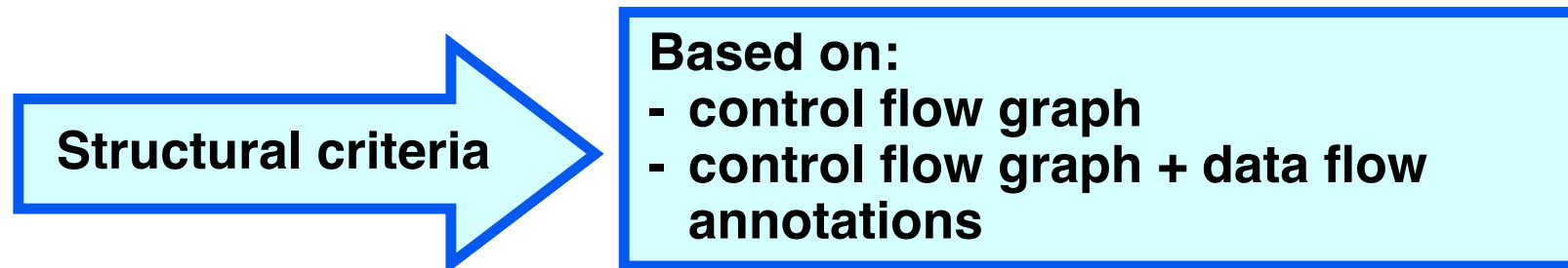
- Fault finding test:** test aimed at uncovering fault
- Conformance test:** functional test aimed at checking whether a software complies with its specification (integrated testing level, required traceability between test data and specification)
- Robustness test:** test aimed at checking the ability of a software to work acceptably in the presence of faults or of stressing environmental conditions
- Regression test :** after software modification, test aimed at checking that the modification has no undesirable consequence

- ➡ **Introduction**
- ➡ **Structural testing**
- ➡ **Functional testing**
- ➡ **Mutation analysis**
- ➡ **Probabilistic generation of test inputs**

Control flow graph

Oriented graph giving a compact view of the program control structure:

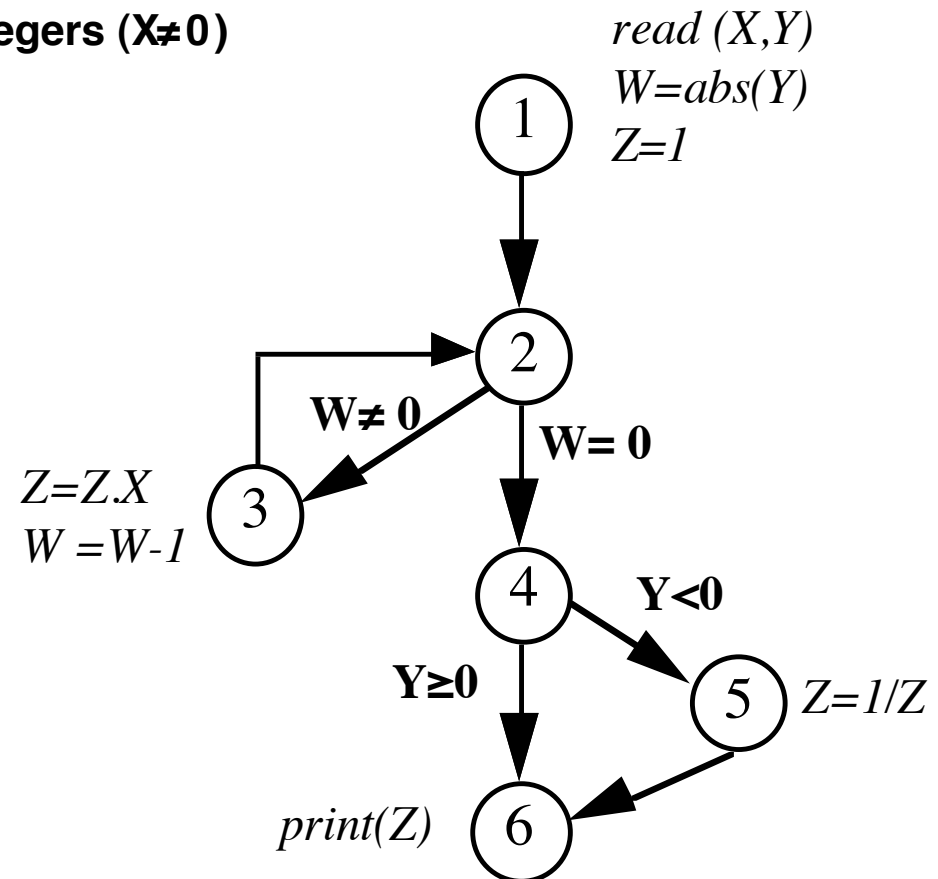
- **built from the program source code**
- **A node = a maximal block of consecutive statements i_1, \dots, i_n**
 - i_1 is the unique access point to the block
 - the statements are always executed in the order i_1, \dots, i_n
 - the block is exited after the execution of i_n
- **edges between nodes = conditional or unconditional branching**



POWER function:

computes $Z = X^Y$, where X and Y are two integers ($X \neq 0$)

```
BEGIN  
  read (X,Y) ;  
  W = abs (Y) ;  
  Z = 1;  
  WHILE (W <> 0) DO  
    Z = Z * X ;  
    W = W-1 ;  
  END  
  IF (Y<0) THEN  
    Z = 1/Z ;  
  END  
  print (Z) ;  
END
```



Program execution = activation of a path in the graph
Structural Criterion: guides the selection of paths

○ All Paths

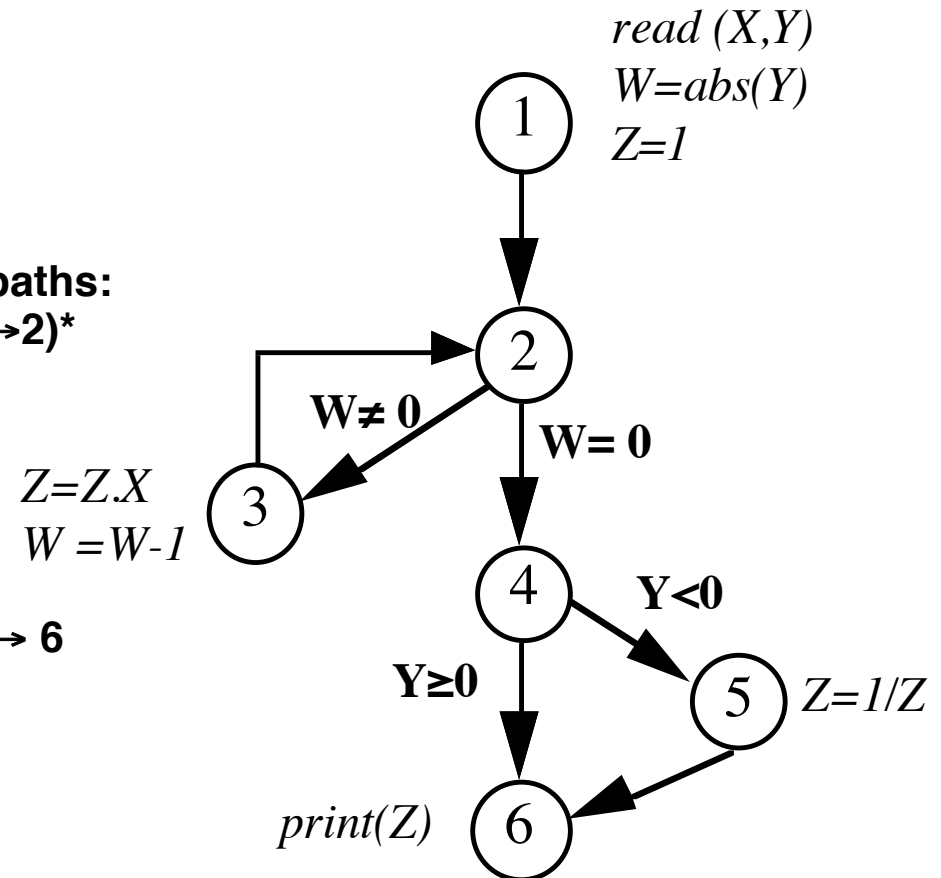
- Non-executable path:
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$
- Infinite (or very large) number of paths:
number of loop iterations $2 \rightarrow (3 \rightarrow 2)^*$
determined by $|Y|$

○ All Branches

- Two executions are sufficient
 $Y < 0 : 1 \rightarrow 2 \rightarrow (3 \rightarrow 2)^+ \rightarrow 4 \rightarrow 5 \rightarrow 6$
 $Y \geq 0 : 1 \rightarrow 2 \rightarrow (3 \rightarrow 2)^* \rightarrow 4 \rightarrow 6$

○ All Statements

- Covered by a single execution
 $Y < 0 : 1 \rightarrow 2 \rightarrow (3 \rightarrow 2)^+ \rightarrow 4 \rightarrow 5 \rightarrow 6$



Other criteria

- **Criteria for covering loops**

- Intermediate between “all paths” and “all branches”
- E.g., pick paths that induce 0, 1 and $n > 1$ loop iterations (you may, or not, consider all subpaths for the loop body at each iteration)

- **Criteria for covering branch predicates**

- Refinement of “all branches” (and also possibly “all paths”) when the branch predicate is a compound expression with Boolean operators.
- Example: $A \wedge (B \vee C)$

Test every possible combination of truth values for conditions A, B et C $\rightarrow 2^3$ cases

Test a subset of combinations such that each condition independently affects the outcome of the decision to False and True

.../...

... Test a subset of combinations such that each condition independently affects the outcome of the decision to F and T...

MC/DC criterion = Modified Condition / Decision Coverage



Principle

A 2 test cases F T
 $\neg A$

$A \wedge B$ 3 test cases FT TF TT

	AB	Dec.	Cond. Affect. Dec.
0	FF	F	–
1	FT	F	A (3)
2	TF	F	B (3)
3	TT	T	A (1), B (2)

$A1 \wedge A2 \dots \wedge An \Rightarrow n+1$ test cases
 single A_i is F (n cases) + case TT...T

$A \vee B$ 3 test cases FF FT TF

	AB	Dec.	Cond. Affect. Dec.
0	FF	F	A (2), B (1)
1	FT	T	B (0)
2	TF	T	A (0)
3	TT	T	–

$A1 \vee A2 \dots \vee An \Rightarrow n+1$ test cases
 single A_i is T (n cases) + case FF...F

Ex : $A \wedge (B \vee C)$

	ABC	Res.	Oper. Affect Res.
1	TTT	T	A (5)
2	TTF	T	A (6), B (4)
3	TFT	T	A (7), C (4)
4	TFF	F	B (2), C (3)
5	FTT	F	A (1)
6	FTF	F	A (2)
7	FFT	F	A (3)
8	FFF	F	—

Take a pair for each operand

A : (1,5) or (2,6) or (3,7)

B : (2,4)

C : (3,4)

Hence two minimal sets for covering the criterion

{2, 3, 4, 6} ou {2, 3, 4, 7}

i.e., 4 cases to test (instead of 8)

Generally: $[n+1, 2n]$ instead of 2^n

Remark: MC/DC can be applied to instructions involving Boolean expressions, in addition to branching conditions

If (A and (B or C))

res := A and (B or C);

MC /DC and coupled conditions

« If a condition appears more than once in a decision, each occurrence is a distinct condition »

- **Example: $(A \wedge B) \vee (A \wedge C)$**

2 occurrences of A: 2 pairs for A

Occurrence 1 -> pair (6,2)

	A	B	C	res
6	T	T	F	T
2	F	T	F	F

Occurrence 2 -> pair (5,1)

	A	B	C	res
5	T	F	T	T
1	F	F	T	F

- **Some cases may be impossible to cover when conditions are not independent**

Example 1: $(A \wedge B) \vee \neg A$

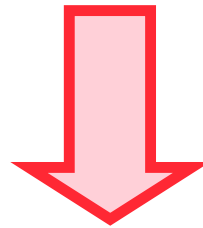
1st occurrence of A cannot affect the decision outcome F

Example 2 : $(A \wedge x \leq y) \vee (x > y-10)$

$x \leq y$ cannot affect the decision outcome F

MC / DC in practice

- **A priori approach via complete truth table often infeasible (e.g., number of operands > 4)**

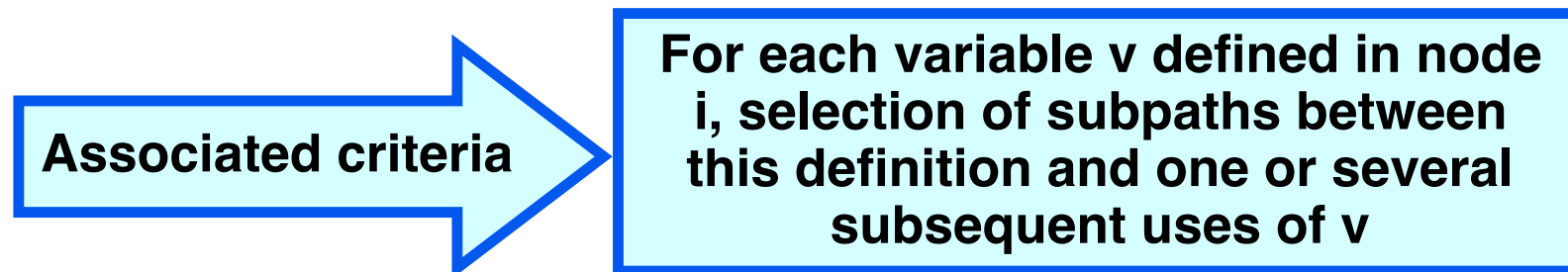


- **A posteriori evaluation of an existing (functional) test set**
 - Coverage analysis tools: IPL Cantata, LDRA Testbed
- **According to results, complement, or not, the test set**

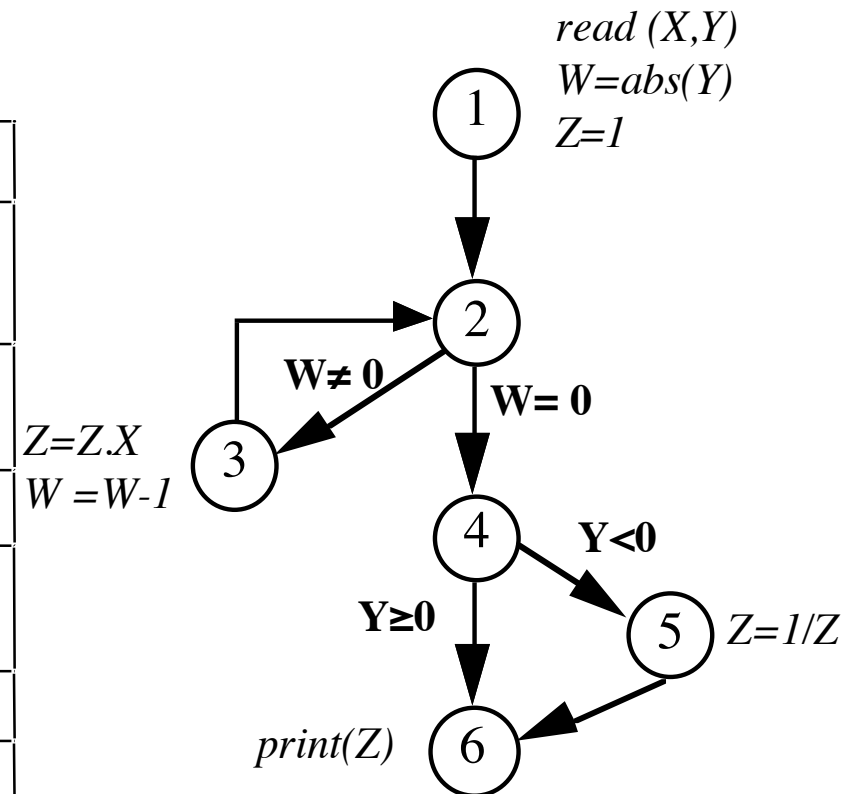
Data flow

Annotating the control graph:

- **Node i**
 - **Def (i)** = set of variables defined at node i , which can be used externally to i
 - **C-use (i)** = set of variables used in a calculus at node i , not defined in i
- **Edge (i, j)**
 - **P-use (i, j)** = set of variables appearing in the predicate conditioning transfer of control from i to j



node i	def(i)	c-use (i)	arc (i,j)	p-use (i,j)
1	X, Y, W, Z		(1,2)	
2			(2,3) (2,4)	W W
3	W, Z	X, W, Z	(3,2)	
4			(4,5) (4,6)	Y Y
5	Z	Z	(5,6)	
6		Z		



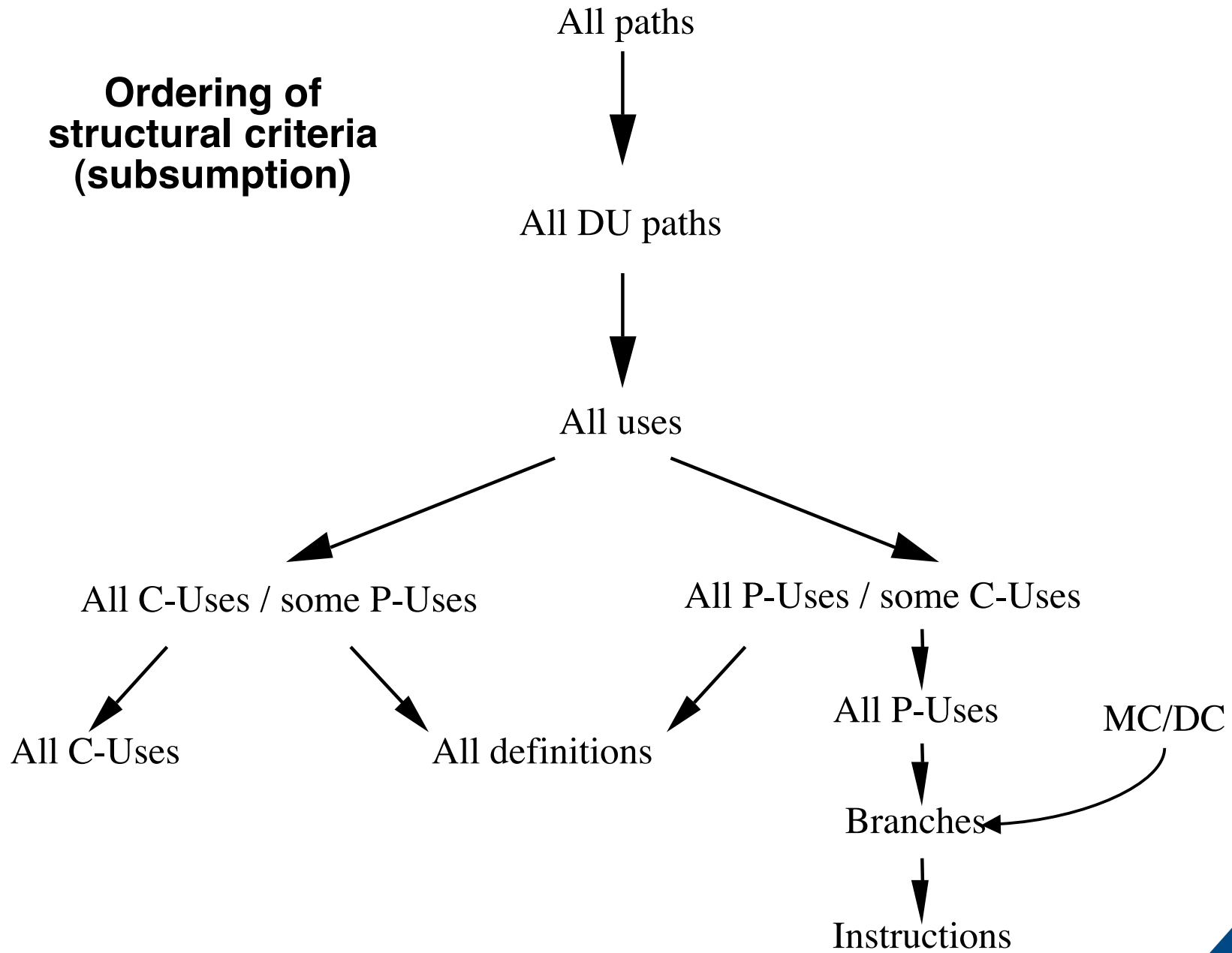
Example : definition of Z at node 1 --> covering all uses?

- use at node 3: test with $|y| > 0$
- use at node 6: test with $y = 0$
- use at node 5 impossible, as path $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$ is infeasible

criteria:

- **All definitions**
 - Selection of a subpath for each variable definition, for some use (equally in a calculus or predicate)
- **All C-uses / some P-Uses (resp. all P-uses / some C-Uses)**
 - Use in calculation (resp. in predicate) is favored
- **All Uses**
 - Selection of a subpath for each use
- **All DU paths**
 - Selection of all possible subpaths without iteration between definition and each use

**Ordering of
structural criteria
(subsumption)**



Structural Criteria – conclusion

- **Criteria defined in a homogeneous framework**
 - Model = control flow graph (+ possibly data flow)
- **Mainly applicable to the first test phases (unit testing, integration of small sub-systems)**
 - Complexity of analysis rapidly grows
 - Note: for integration testing, a more abstract graph may be used (call graph)
- **Tool support (except for data-flow)**
 - Automated extraction of control-flow graph, coverage analysis
- **Structural coverage is required by standards**
 - Typically: 100% branch coverage
 - Can be more stringent: MC/DC for DO-178B

- ➡ **Introduction**
- ➡ **Structural testing**
- ➡ **Functional testing**
- ➡ **Mutation analysis**
- ➡ **Probabilistic generation of test inputs**

Equivalence classes + boundary values

Principle

Partition the input domain into equivalence classes to be covered

Classes determined from the functional requirements (set of values for which functional behavior is the same),
and/or from the data types (e.g., for *int*, positive, negative)

Consider both valid and invalid classes (robustness testing)

Identify boundary values for dedicated tests

E.g., -1, 0, 1, +/- MAXINT

Example

“The price entered by the operator must be a strictly positive integer.
If $\text{price} \leq 99\text{€}$, then ... From 100€ and above, the processing ...”

Valid classes:

$1 \leq \text{price} \leq 99$

$100 \leq \text{price}$

+ possibly:

Invalid class:

$\text{price} \leq 0$

$\text{price} = \text{real number}$

$\text{price} = \text{string}$

Boundary values:

$\text{price} = -1, 0, 1, 98, 99, 100,$

$101, \text{+/- MAXINT}$

nothing entered (\leftarrow)

- **Informal approach, as based on natural language specification**
 - Differing partitions and limit values can be obtained from same specification analysis
- **Analysis granularity?**
 - **Separate analysis of input variables \Rightarrow considering all case combinations?**

For <i>price</i> input variable	$1 \leq \text{price} \leq 99$	$100 \leq \text{price}$
For <i>in_stock</i> input variable	$\text{in_stock} = \text{TRUE}$	$\text{in_stock} = \text{FALSE}$
 - **Class defined by composite predicate \Rightarrow breaking up in disjoint cases?**

discount = 10% or discount = 20%
 - **Classes with non-empty intersection \Rightarrow breaking up in disjoint cases?**

$1 \leq \text{price} \leq 99$ $\text{prix} * (1 - \text{discount}) \leq 50$
- ☹ **Rapid explosion of the number of created classes!**
- **Examples of selective choice strategies aimed at covering the classes**
 - Selecting one test input for each class, without considering possible intersections
 - Minimizing the number of test inputs while favoring the patterns covering several valid classes; contrarily, testing separately each invalid class
 - Pairwise strategies for covering combinations of values from classes

Decision Table

		rule 1	rule 2	rule 3	rule 4
LIST OF CONDITIONS	C 1	TRUE	TRUE	FALSE	FALSE
	C 2	X	TRUE	FALSE	FALSE
	C 3	TRUE	FALSE	FALSE	TRUE
LIST OF ACTIONS	A 1	YES	YES	NO	NO
	A 2	YES	NO	YES	YES

For a combinatorial function

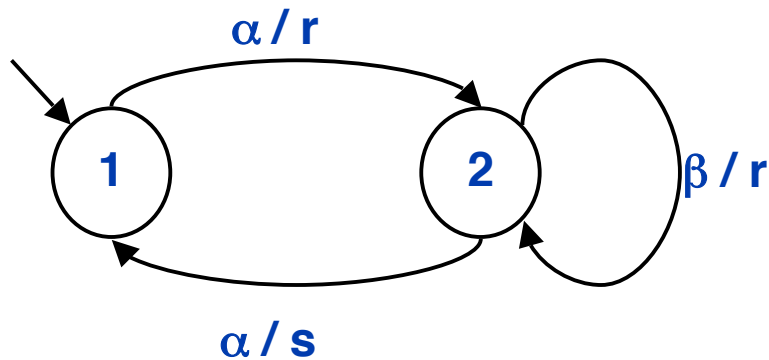
- **C_i : input conditions**
A_i : disjoint actions, order in which they are executed does not matter
- **completeness and consistency of the rules (exactly one rule is eligible)**
➡ document the impossible cases

👉 Rule coverage = 1 test case for each rule

Finite State Machine

For a sequential function

- S : finite set of states
- S_0 : initial state
- Σ : finite input alphabet
- Ω : finite output alphabet
- δ : transition relation, $\delta : S \times \Sigma \rightarrow S$
- λ : output relation, $\lambda : S \times \Sigma \rightarrow \Omega$



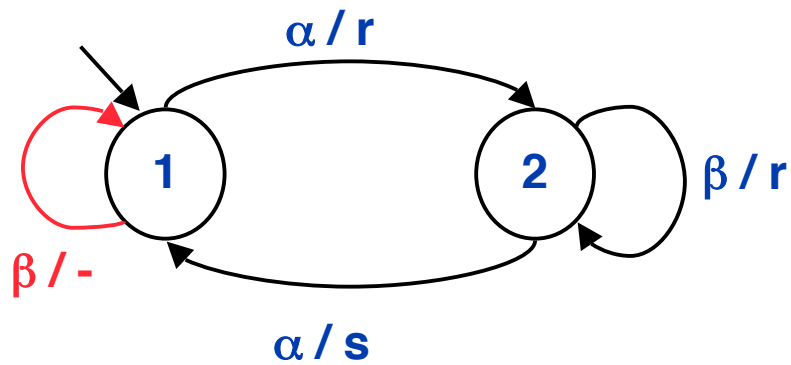
Graphical representation

	α	β
1	2/r	?
2	1/s	2/r

Tabular representation

Preliminaries:

- Check completeness. E.g., add self-loop ' $\beta/-$ ' on State 1
- Deterministic? Minimal? Strongly connected?



Graphical representation

	α	β
1	2/r	1/-
2	1/s	2/r

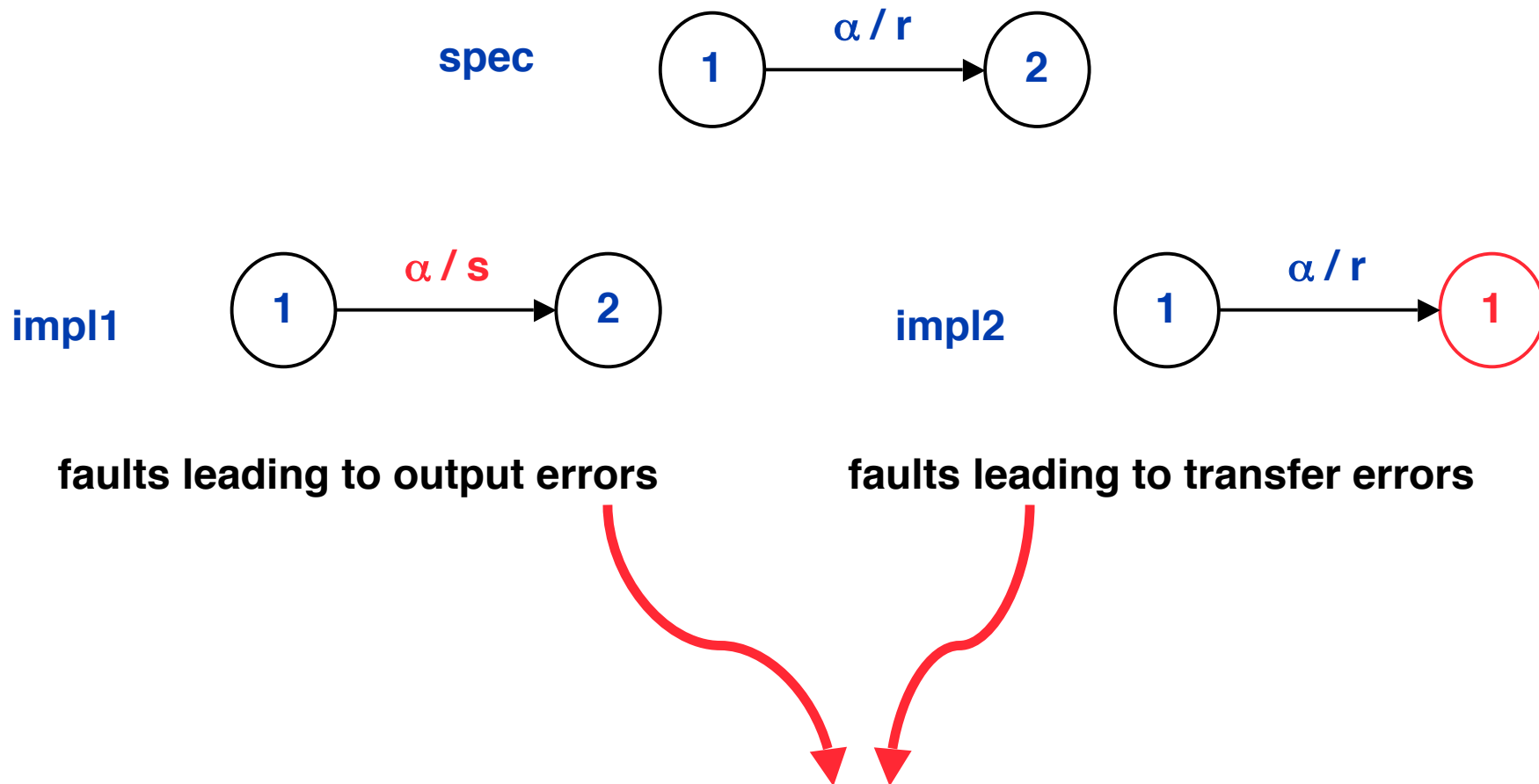
Tabular representation

⇒ coverage of states, transitions, switches (pairs of transitions)

- Testing wrt original machine or **completed** one
Impact on the input domain + oracle
- controllability: reach State i to test transition $i \rightarrow j$?
Availability of a 'reset' function in the test driver

⇒ Other strategies based on fault models

... / ...



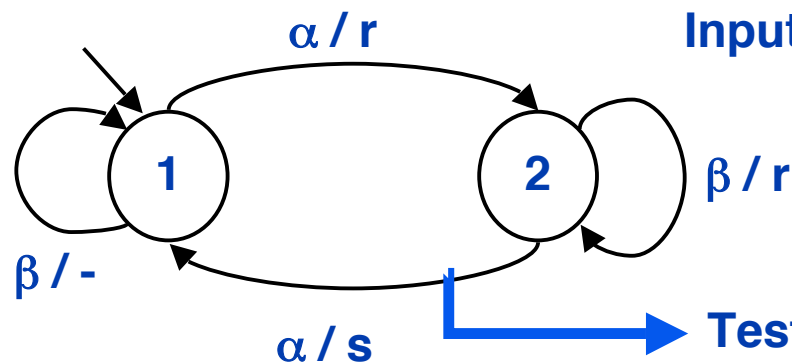
➔ **Testing all transitions + checking the arrival state**

- **observability: observing state j?**
Existence of a 'status' function?

Generally, direct observation of state i is impossible
 Selective choice strategy often used:

$$T_{i,j} = \text{preamble}_i \cdot e_{ij} \cdot \text{SC}_j$$

With: T_{ij} = test sequence for transition $i \rightarrow j$
 preamble_i = input sequence starting at initial state, and arriving at i
 e_{ij} = input activating transition $i \rightarrow j$
 SC_j = characterising sequence for state j



Input α enabling to distinguish between ① and ②:
 ① \rightarrow r output ② \rightarrow s output

Testing transition ② $\xrightarrow{\alpha / s}$ ① via sequence:

Reset . α . α . α

Generally, direct observation of state j is impossible
 Selective choice strategy often used:

$$T_{i,j} = \text{preamble}_i \cdot e_{ij} \cdot \text{SC}_j$$

With: T_{ij} = test sequence for transition $i \rightarrow j$
 preamble_i = input sequence starting at initial state,
 and arriving at i
 e_{ij} = input activating transition $i \rightarrow j$
 SC_j = characterising sequence for state j



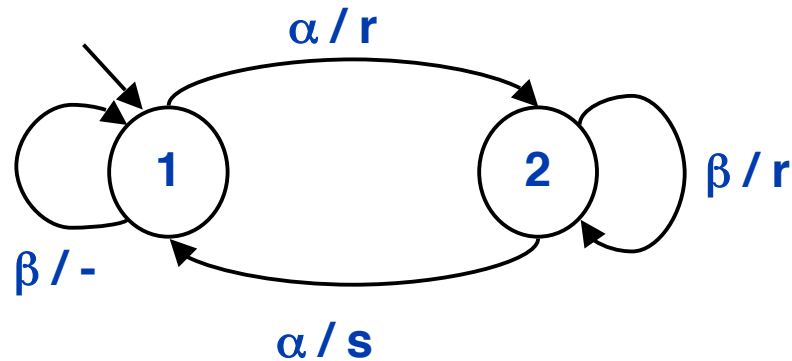
Do not always exist {

- Distinguishing sequence DS (same $\forall j$)
- Sequence UIO (unique for each j)

Exists for each minimal and complete MEF {

- Set W (set of sequences enabling to distinguish states 2 by 2)

Example: Cho's W Method

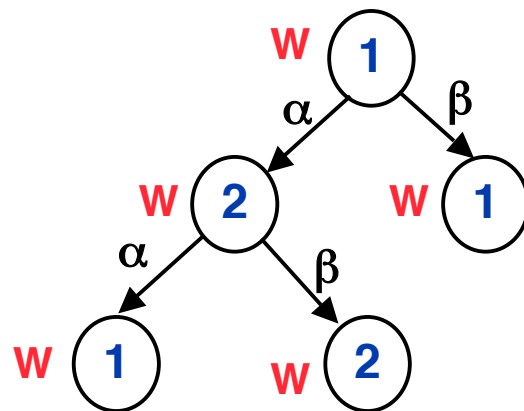


$[1\ 2]$
 \downarrow
 $[1]\ [2]$

$W = \{\alpha\}$
 (Here, DS)

General case

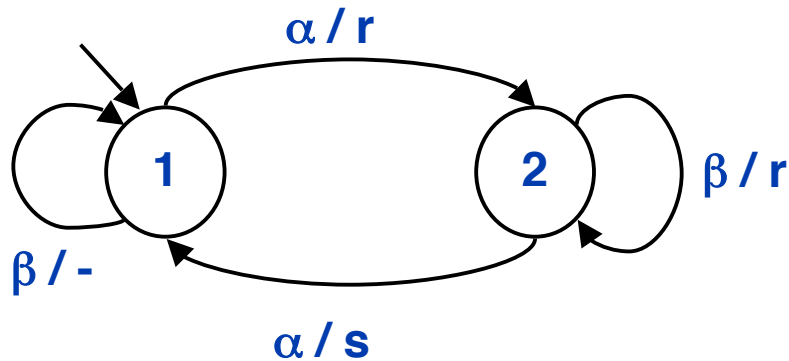
$[1\ 2\ \dots\ n]$
 \downarrow $W = \{\alpha\}$
 $[1\ 3]\ [2, 4\ \dots\ n]$
 \downarrow $W = \{\alpha, \beta \cdot \beta\}$
 $[1]\ [3]\ [2, \dots]\ [4]\ [5, \dots]$



Test tree

Reset $\cdot \alpha$
 Reset $\cdot \beta \cdot \alpha$
 Reset $\cdot \alpha \cdot \alpha$
 Reset $\cdot \alpha \cdot \alpha \cdot \alpha$
 Reset $\cdot \alpha \cdot \beta \cdot \alpha$

Test sequences



W method

Reset · α
 ✓ Reset · β · α
 Reset · α · α
 ✓ Reset · α · α · α
 ✓ Reset · α · β · α

Transition tour

Reset · α · β · α · β

Rapid explosion of the test size

W method:

For a completely specified, minimal, deterministic FSM of n states

If implementation is also deterministic, with n states at most, the driver implements correctly the Reset,

Then the W method guarantees to reveal faults producing output errors (according to the output alphabet) and transfer errors

Strong assumptions

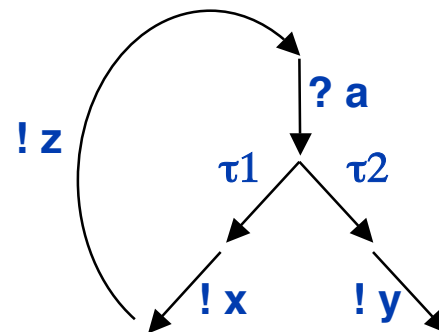
...

Transition systems

LTS (Labelled transition system) = low level formalism for describing process systems / communicating automata

Example: specifications in LOTOS, SDL, Estelle, ... can be translated in LTS

- **Basic LTS: input and output events are not distinguished**
⇒ **IOLTS (Input/Output LTS) for testing**



Input event: a
Output event: x, y, z
Internal actions: τ_1, τ_2

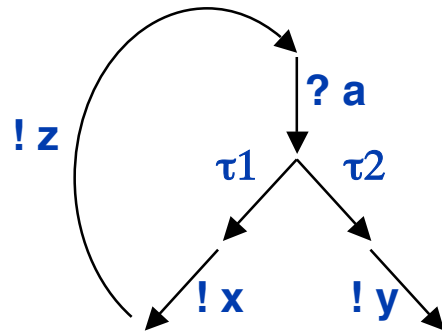
non-determinism, quiescence

- **Test approaches referring to relations between LTSs**
Chosen relation = conformance relation between spec and impl

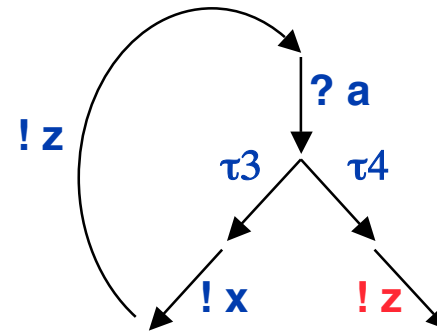
Example: ioco relation

Implementation not in conformity if it can exhibit outputs or quiescences not present in specification

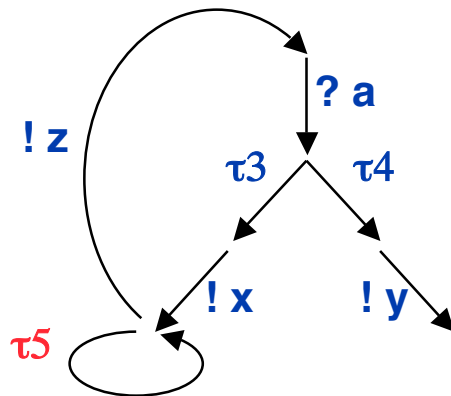
Spec



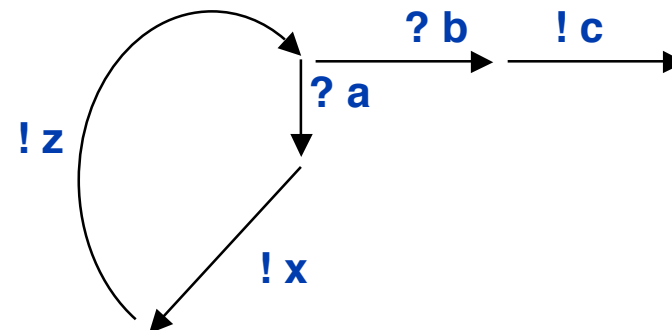
Impl1 **not in conformity**



Impl2 **not in conformity**



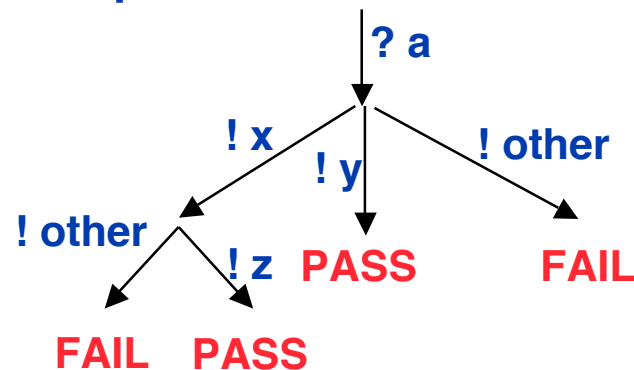
Impl3 in conformity



Test cases

Test case = IOLTS with specific trap states (Pass, Fail, Inconclusive)
 --> incorporates inputs, expected outputs, and verdict

Example:



- enables non conformity of impl1 to be uncovered
Observation of !z after ?a
- enables non conformity of impl2 to be uncovered
Observation of a quiescence after ?a !x

Test case automatically synthesized from a test purpose

Test purpose
=IOLTS



Spec
=IOLTS



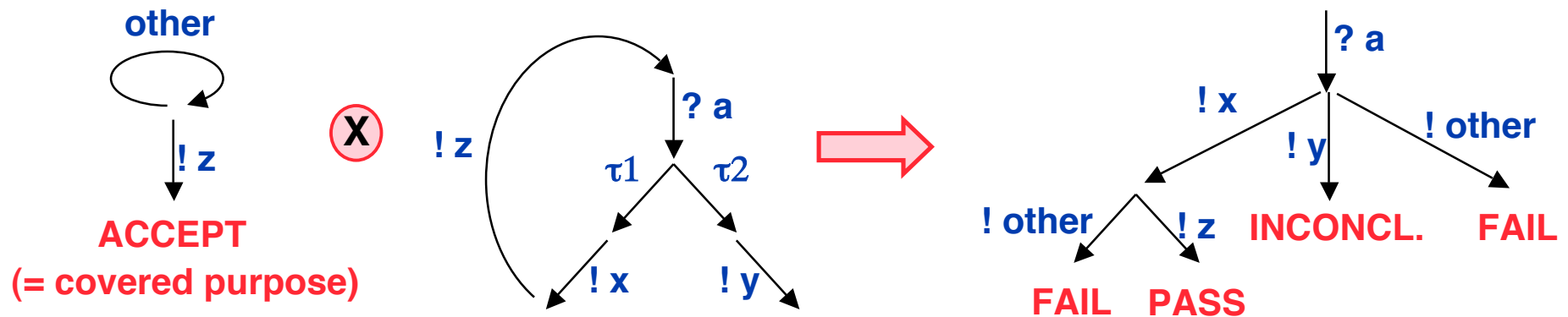
Test case
=IOLTS

Test purpose

- handwritten
- automatically generated from a specification coverage criterion

Example: transition coverage of the SDL model

Example: purpose = produce !z



(Actually, the mirror image of this)

Functional Testing

- **No homogeneous framework, methods depend on the used model**
 - Equivalence classes
 - Decision tables
 - Finite state machines
 - Statecharts
 - SDL
 - Logic-based specifications (algebraic, model-oriented)
 - ...
- **Model abstraction level & criterion stringency depend on the complexity (integration level) of the tested software**
- **Formal models used for specification and design \Rightarrow makes it possible to (partially) automate testing = input generation, oracle**

- ➡ **Introduction**
- ➡ **Structural testing**
- ➡ **Functional testing**
- ➡ **Mutation analysis**
- ➡ **Probabilistic generation of test inputs**

Assessing the fault-revealing power of test methods?

- **Feedback from past projects**
 - Assessment wrt real faults
 - But reduced sample of faults
- **Mutation analysis = controlled experiments**
 - Assessment wrt seeded faults
 - Large sample of faults

Mutation analysis (1)

- ☞ introduction of a simple fault (= mutation)

“C” → “C +1” “true” → “false” “+” → “-” “<” → “≤” “low” → “mid”

- ☞ execution of each mutant with test set T
 - killed (fault revealed by the test set)
 - alive (fault not revealed)

- ☞ measurement = mutation score

Number of mutants killed by T

Total number of mutants

- ☞ mutations are syntactically not representative of development faults but produce similar errors
(it is as difficult to reveal mutations as to reveal real faults)

- ☞ Tools

- Mothra (Georgia Inst. of Techn.): Fortran
- Sesame (LAAS): C, Pascal, Assembly, Lustre
- JavaMut (LAAS): Java
- *and many others...*

Mutation analysis (2)

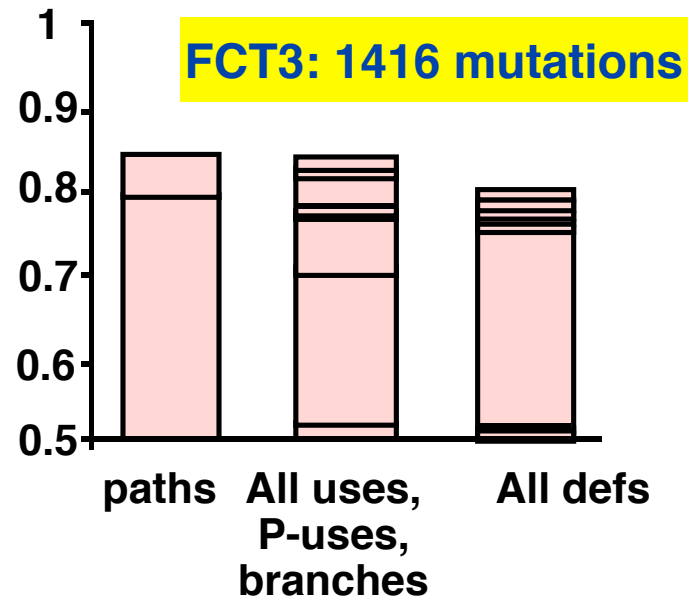
- ☺ **Comparing test methods (academic research)**
- ☺ **Identifying imperfections of a structural or functional test set**
--> complement test set with additional cases
- ☺ **Evaluating software propensity to mask errors**
--> locally more stringent test

But ...

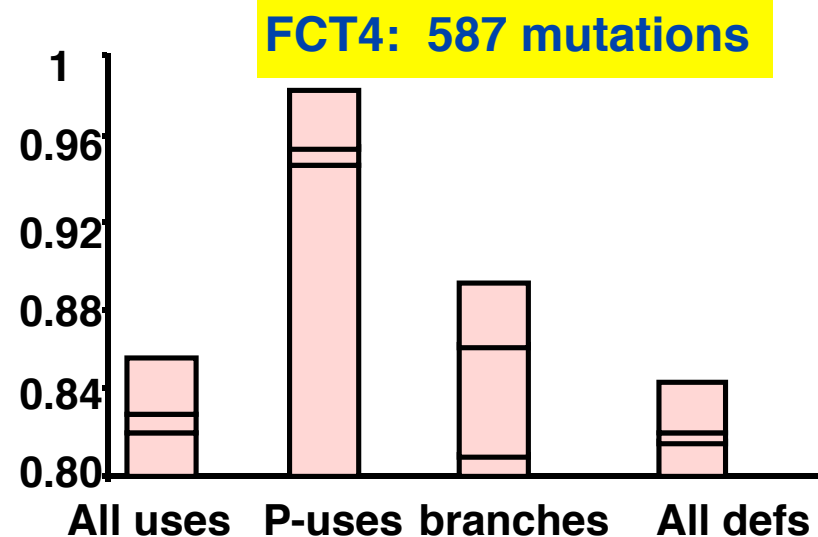
- ☹ **Explosion of the number of mutants**
- ☹ **Identification of equivalent mutants partly by hand**
equivalent mutant = mutant which cannot be distinguished from the original programme by any test input

Effectiveness of structural criteria: some experimental results

Mutation score



Mutation score



- All paths: score < 100%
- For a given criterion, strong impact of the chosen input values
- Criterion stringency → no guarantee

Imperfection of test criteria

☞ Deterministic approach: criterion refinement

Examples :

instructions → branches → paths → paths + decomposition of branch conditions

states → transitions → W method → W+1 method (number of implemented states \leq number of specified states +1) → ... → W+n method

☺ exhaustiveness according to fault assumptions

☹ explosion of test size in order to account for weaker assumptions

☞ Probabilistic approach

Random, less focused, selection: exhaustiveness according to fault assumptions not searched for, reasoning in terms of failure rate according to a sampling profile.

.../...

Remark: increase of the test size in both cases

⇒ **Necessity to automate input selection and oracle procedure**

- ➡ **Introduction**
- ➡ **Structural testing**
- ➡ **Functional testing**
- ➡ **Mutation analysis**
- ➡ **Probabilistic generation of test inputs**

Probabilistic approaches

Random testing

Uniform distribution over the input domain

☺ Easy to implement

☹ “blind” selection, usually inefficient... usually not recommended!

Operational testing

Input distribution is representative of operational usage

☺ Removal of faults that will have the greatest impact on reliability + software reliability assessment

☹ inadequate for revealing, or assessing the consequences of, faults that induce small failure rates (e.g., $< 10^{-4}/h$)

Statistical testing = criterion + random selection

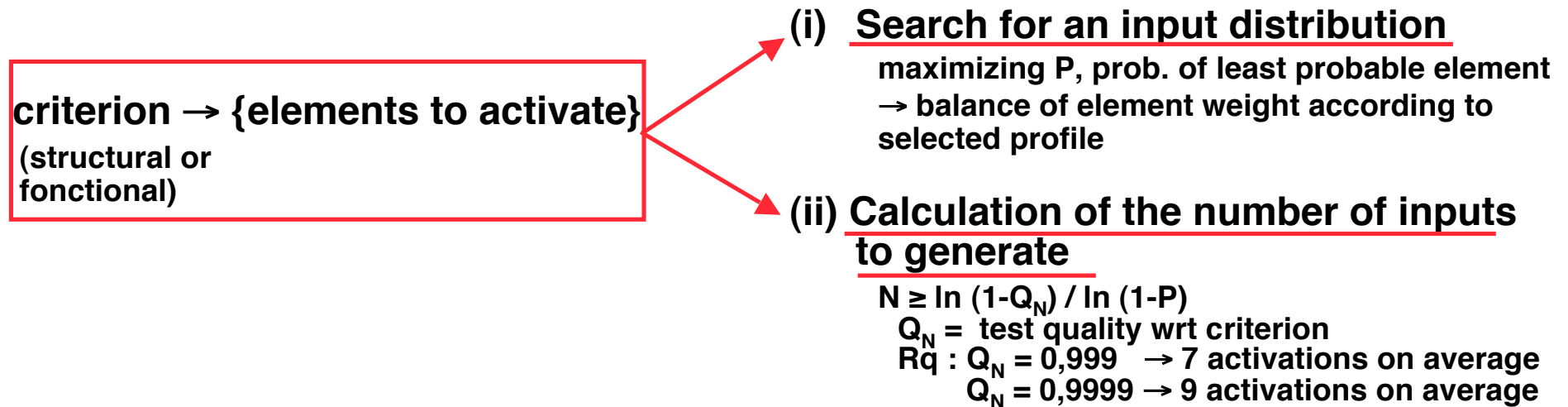
Imperfect but relevant
information

compensates imperfection
of criterion

Operational testing

- **Population of users \Rightarrow test according to several operational profiles**
- **Definition of operational profile(s): functional modeling**
 - Identifying operating modes, functions which can be activated in a mode, input classes
 - Quantifying probabilities associated to the model according to operating data for similar systems, or to projected estimates
 - \Rightarrow introducing measurement capabilities in the system (« log » files)
- **Some figures (Source: AT&T)**
 - Operational profile(s) definition = 1 person.month for a project involving 10 developers, 100 KLOC, 18 months
 - Definity project: duration of system test %2, maintenance costs %10

Criterion-based statistical testing



Structural statistical testing

Probabilistic analysis of control graph and data flow

Functional statistical testing

Probabilistic analysis of behavior models

Finite state machine, decision table, Statechart

Example



Analytical techniques

$C = \{\text{paths}\} \rightarrow Sc = \{k1, k2\}$



$p1 = \text{Prob}[0 \leq Y \leq 99]$

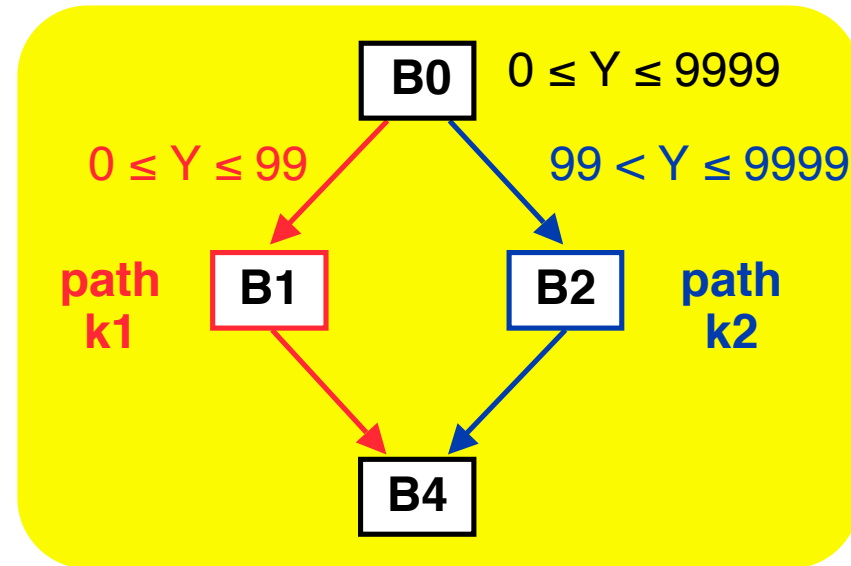
$p2 = \text{Prob}[99 < Y \leq 9999]$



$p1 = p2 = 0.5$

Structural distribution / C

$Q_N = 0,999 \rightarrow N = 10$ inputs to generate



Empirical techniques

Successive refinements of an initial distribution

results for 4 unit functions and 2816 mutations :

TEST TYPE	LIVE MUTANTS	MUTATION SCORE
Structural deterministic	from 312 to 405	[85,6% – 88,9%]
Uniform random	from 278 to 687	[75,6% – 90,1%]
Structural statistical	6	99,8%

Typical cases for limit value testing

Contribution of probabilistic approach?

Compensates imperfection of structural criteria

Contribution of criterion to efficiency of random inputs?

Adequate test profiles

Functional statistical testing: mastering complexity?

- adoption of weak criteria (e.g., states,...) and possibly definition of several profiles
- high test quality (e.g., $q_N = 0.9999$)

Example : component from a nuclear control system

Functional modeling: hierarchy of Statecharts

⇒ Statistical testing according to two complementary profiles $N = 85 + 356 = 441$

Efficiency wrt actual faults?

« Student » version: 12 faults (A, ..., L)

« industrial » version: 1 fault Z (hardware compensation)

→ programming:	A, G, J
→ understanding:	B, ..., F, I
→ initialisation:	H, K, L, Z

Efficiency wrt injected faults?

« Student » version: mutation score = 99.8 - 100%

« industrial » version: mutation score = 93 - 96.1%

**Necessity of initialisation
process specific test**

General conclusion

- **Numerous test methods**
 - General approach: take a (structural or functional) model and define model elements to be covered
- **Choice**
 - Depends on available models and their complexity
 - Complementarity of “global” coverage and testing of specific cases (boundary values, transient modes, ...)
 - Deterministic or probabilistic selection
- **(Semi-)automation of testing is highly recommended**
 - B. Beizer : “About 5% of the software development budget should be allocated to test tool building and/or buying ”
- **Beyond this introductory lecture...**
 - Not covered here: specificities wrt testing OO, concurrent, distributed, real-time software systems