TEST SETS GENERATION
FROM
ALGEBRAIC SPECIFICATIONS
USING
LOGIC PROGRAMMING

L. Bougé
N. Choquet
L. Fribourg
M.-C. Gaudel

Laboratoires de Marcoussis
Centre de Recherches de la CGE
funded by CIT-Alcatel
THE PROBLEM

FUNCTIONAL OR BLACK-BOX TESTING

* TEST of \( \langle X, Y \rangle \) AGAINST SOME SPECIFICATION

PROGRAM WHITE-BOX TESTING

*Implementation dependent
*Test of each possible path
*Search for specific kinds of errors

NON-REGRESSION TESTING

CHECK THAT \( \text{RELEASE}^{n+1} \)

"INCLUDES" \( \text{RELEASE}^n \)

implementation independent
expensive

-SPECIFIC TEST DATA SETS

-SYSTEMATIC CONSTRUCTION AND MANAGEMENT
AIMS OF THE STUDY

* TO GIVE RIGOUROUS BASIS TO THE TESTING PROCESS:

TO STUDY THE COMPLEMENTARITY BETWEEN TESTING AND PROVING;

TO MAKE (EXPlicit) ALL THE ASSUMPTIONS WHICH ARE GENERALLY (IMPLICIT) WHEN SOMEONE SAYS:

"TEST T IS SUCCESSFUL \( \Rightarrow \) PROGRAM P IS CORRECT"

* TO PROVIDE METHODS FOR TEST SETS CONSTRUCTION FROM FORMAL SPECIFICATIONS

PLAN

I) BASIC DEFINITIONS ON TESTING

II) APPLICATION TO ALGEBRAIC SPECIFICATIONS

III) WHY LOGIC PROGRAMMING IS AN APPROPRIATE TOOL
THE PROBLEM

Does a Program P satisfy a Specification S?

i.e.

Does a $\Sigma$-algebra $X$ satisfy a $\Sigma$-axiom $A$?

BASIC IDEA

IF A IS AN EQUATION:

lhs($x$) - rhs($x$)

- instantiate $x$ by some constant $\Sigma$-term;
- compute both sides in $X$;
- decide if the results are equal;
- start again.

TESTING PROCESS DIAGRAM

$\Sigma$-ALGEBRA

+ REQUIRED PROPERTIES

(AXIOMS)
\[ \Sigma\text{-ALGEBRA} + \text{REQUIRED PROPERTIES (AXIOMS)} \]

\[ \text{TEST SET} \]

\[ \text{TESTING CONTEXT} \]

\[ \text{COLLECTION OF TEST SETS} \]

\[ \Sigma\text{-ALGEBRA} \]

\[ \text{TEST SET} \]

\[ \text{TESTING CONTEXT} \]

\[ \text{COLLECTION OF TEST SETS} \]
Property to be tested (formula)

Class of $\Sigma$-algebras

(P belongs to (S))

Testing Hypotheses

$\Sigma = \{ \text{emptyq, append, remove, first} \}$

$P$ = a piece of software implementing functions of these names

$A = \{ q \cdot \text{emptyq} \Rightarrow$

$\text{first}(\text{append}(q,x))=\text{first}(q) \}$

Example of a collection of test data sets

$T_p = ( \text{a}_i : i=1, \ldots, f(p))$

where $a_i$ are closed instantiations of $A$
SUMMARY

DOES THE \( \Sigma \)-ALGEBRA \( P \) SATISFY THE EQUATION \( \text{lhs}(x) = \text{rhs}(x) \)?

BUILD \( (T_n)_{n \in \mathbb{N}} \)

SELECT \( T_P \) - ( \( \text{lhs}(t_1) - \text{rhs}(t_1) \) / \( \vdash \ldots f(p) \))

FOR EACH \( t_1 \) COMPUTE \( \text{lhs}(t_1) \) and \( \text{rhs}(t_1) \)

DECIDE IF THE RESULTS ARE EQUAL

QUESTION: IS THE CONCLUSION SENSIBLE?

OR

WHAT ARE THE REQUIREMENTS ON \( (T_n) \)

TO GET A MEANINGFUL CONCLUSION?

RELIABILITY

A collection of test sets is said to be reliable if a test set of higher index is “better” than a test set of lower index whatever potential \( \Sigma \)-algebra is considered.

\[
\forall n \in \mathbb{N}, (H \cup T_{n+1}) \vdash T_n
\]

Property slightly weaker than Goodenough and Gerhart’s one:

\( \Rightarrow \) asymptotic reliability.

VALIDITY

Any incorrect behavior will be revealed by some test data in some \( T_n \), i.e.

\[
(H \cup (\cup_n T_n)) \vdash A
\]

If testing is successful using all test sets \( T_n \) then the algebra fulfills the required properties. A collection which satisfies this property is said to be

\( \Rightarrow \) asymptotically valid.

LACK OF BIAS.

Any correct algebra should pass any test set \( T_n \)

Converse of validity:

\[
\forall n \in \mathbb{N}, (H \cup A) \vdash T_n
\]

If an experiment using the test set \( T_P \) selected from the collection \( (T_n) \) fails, then the algebra does not satisfy the axioms.
limits of this approach

\[ S = \mathbb{N}, \quad L = \{ P \}, \quad A = \{ \exists x \ P(x) \} \]

(E) \quad P \circ \text{ is computable}

Let us suppose that \( (T_n) \) is a not biased test net \[ \implies \forall S', S' = A \implies S' = T_n \ \forall n \in \mathbb{N} \]

all non-logical axioms of \( T_n \) can be reduced
\[ P(n_1) \lor \ldots \lor P(n_p) \lor \neg P(n_1) \ldots \lor \neg P(m_q) \]

let us consider \( S \) s.t.
\[ P \circ = (x = m_1) \lor \ldots \lor (x = m_q) \]
\[ S = A \quad \text{and} \quad S \not\models T_n \]

Thus the axioms are:
\[ P(n_1) \lor \ldots \lor P(n_p) \]

but with
\[ P \circ = (x = m) \quad m \neq n_1 \ldots \quad m \neq n_2 \]
\[ S = A \quad \text{and} \quad S \not\models T_n \]
\[ ... \quad (T_n) \text{ is empty} \]

The only test set which is not biased is the empty one! (When there is an existential quantifier in \( A \))

EQUATIONAL CASE

THE SPECIFICATION IS HIERARCHICAL EQUATIONAL
(i.e there is a sort of interest in \( \Sigma \)
there are predefined sorts
axioms are equations)

FOR EACH AXIOM \( A \) OF THE SPECIFICATION

CONSIDER \( (T_n) \) obtained by:

* IF \( x \) IS A VARIABLE OF THE SORT OF INTEREST, INSTANTIATE IT BY ALL THE TERMS OF THIS SORT, OF SIZE LESS THAN \( n \), WITHOUT VARIABLES OF THE SORT OF INTEREST;

* IN THE RESULTING SET OF EQUATIONS INSTANTIATE THE VARIABLES OF PREDEFINED SortS BY RANDOM TERMS.

WHAT ARE THE HYPOTHESES?
REGULARITY HYPOTHESIS

∀x (\text{complexity}(x) \leq k \Rightarrow t(x)=t'(x)) \Rightarrow ∀x (t(x)=t'(x))

\{x \mid \text{complexity}(x) \leq n\} \text{ finite for all integer } n \Rightarrow T_n = \{ t(x)=t'(x) \mid \text{complexity}(x) \leq n\}

is an acceptable collection of test sets.

COMPLEXITY = \text{length of a representative } \Sigma\text{-term denoting an object (computation complexity)}

UNIFORMITY HYPOTHESES

QUITE COMMON in TESTING METHODOLOGIES

\iffalse
Example:
\begin{align*}
\leq (x,y) &= \text{true} \land \text{sorted (append (e,x),y)} = \text{true} \\
\text{insert (append (e,x),y)} &= \text{append (append (e,x),y)}
\end{align*}
\fi

\iffalse
\begin{align*}
Pb: & \text{ provide relevant values for } e, x, y. \\
Note: & \text{ the uniformity hypothesis on predefined sorts is no more valid!}
\end{align*}
\fi

\iffalse
2nd (more realistic) case
Axioms are conditional
\fi

Example:
\begin{align*}
\leq (x,y) &= \text{true} \land \text{sorted (append (e,x),y)} = \text{true} \\
\text{insert (append (e,x),y)} &= \text{append (append (e,x),y)}
\end{align*}

Pb: provide relevant values for e, x, y.

Note: The uniformity hypothesis on predefined sorts is no more valid!
1)- CASE OF EQUATIONAL AXIOMS

Regularity hypothesis for the sort of interest
Uniformity hypotheses for lower sorts

2)- CASE OF CONDITIONAL AXIOMS

Finite decomposition hypothesis

∀ x ∈ D, a(x) = b(x) => t(x) = t'(x)

<=>

∀ x ∈ D1, t(x) = t'(x)

with D1 = {x | a(x) = b(x)}

D1 may be an union, for instance:

∀ x ∈ N, or(le(x,2),le(5,x))=true => t(x) = t'(x)

∀ x ∈ D11, t(x) = t'(x)

∀ x ∈ D12, t(x) = t'(x)

where D11 = {n | n≤2} and D12 = {n | n>5}

INFINITY OF SUCH SUBDOMAINS => REGULARITY-LIKE HYPOTHESES:

∀ i, 1≤i≤k, ∀ x ∈ D1i, t(x) = t'(x) => ∀ x ∈ D1, t(x) = t'(x)

FINITE DECOMPOSITION HYPOTHESES

Method

Transform any conditional axiom

[f. i. a(x) = true => f(x) = g(x)]

into an equivalent set of equational axioms:

{ f(t) = g(t') | a(t) = true }

and apply the previous method.

G general, complete procedure for generating terms satisfying boolean equations

E (E-Axioms) \rightarrow G \rightarrow s = \{ s_1, \ldots, s_n \ldots \}$

P (Premises) \rightarrow G \rightarrow \sigma = \{ s_1 \ldots s_n \ldots \}$

$E \models \sigma \leftarrow A$

The axiom becomes:

{ f(\sigma_i(x)) = g(\sigma_i(x)) | \sigma_i \in \sigma \}$
fig. 3 Translation of the queue specification into PROLOG

C1: isempty(emptyq,true).
C2: isempty(append(Q,I),false).
C3: remove(emptyq,emptyq).
C4: remove(append(Q,I),emptyq):-isempty(Q,true).
C5: remove(append(Q,I),append(Q',I)):-isempty(Q,false),remove(Q,Q').
C6: first(append(Q,I),I):-isempty(append(Q,I),false),isempty(Q,true).
C7: first(append(Q,I),J):-isempty(append(Q,I),false),isempty(Q,false),first(Q,J).
A1: no constraint, no variable.

A2: no constraint, regularity hypothesis for Q, uniformity hypothesis for I:

\[
\begin{align*}
Q_0 &= \text{emptyq}, I_0 = c_{11}^0, \\
Q_i &= \text{append}(\text{emptyq}, c_{2i}^0), I_i = c_{2i}^0, \\
&\vdots \\
Q_n &= \text{append}^{n}(\text{emptyq}, c_{2n}^0), c_{2n}^0), I_n = c_{2n}^n.
\end{align*}
\]

A3: no constraint, no variable.

A4: a constraint on Q: isempty(Q)=true solved by Q=isempty; no constraint on I, uniformity hypothesis for I.

Q = emptyq, I = c^4.

A5: a constraint on Q: isempty(Q)=false solved by Q=append(X,I); no constraint on X, regularity hypothesis on X; no constraint on I and J, uniformity hypothesis on I and J.

\[
\begin{align*}
Q_0 &= \text{append}(\text{emptyq}, c_{11}^0), I_0 = c_{12}^0, \\
Q_i &= \text{append}(\text{emptyq}, c_{2i}^0), c_{2i}^0), I_i = c_{2i}^0, \\
&\vdots \\
Q_n &= \text{append}^{n}(\text{emptyq}, c_{2n}^0), c_{2n}^0), I_n = c_{2n}^n.
\end{align*}
\]

A6: constraints on Q and I: isempty(Q)=true and isempty(append(Q,I))=false, solved with Q=emptyq, for any I (uniformity hypothesis). Q and I are instantiated with similar values than for A4.

A7: constraints on Q and I: isempty(Q)=false and isempty(append(Q,I))=false, solved with Q=append(X,I), for any I. Q and I are instantiated with similar values than for A5.

fig. 4 Instantiation sets generated for the queue specification

specif sorted-list =
use bool, int
sort list;
opendings
operations

el : list * int -> list; /* empty-list constructor */
ap : list * int -> list; /* append constructor */
sorted : list -> bool; /* defined for a sorted list */
insert : list * int -> list;

preconditions
/* The operation insert is used to insert an integer in a sorted list and to get as a result a sorted list. */
pre(insert, L, X) = (sorted(L) = true)

axioms
A1: sorted(el) = true;
A2: sorted(ap(el, X)) = true;
A3: le(X, Y) = true => sorted(ap(ap(L, X), Y)) = sorted(ap(L, X));
A4: le(X, Y) = false => sorted(ap(ap(L, X), Y)) = false;
A5: insert(el, X) = ap(el, X);
A6: le(X, Y) = true => insert(ap(L, X), Y) = ap(ap(L, X), Y);
A7: le(X, Y) = false => insert(ap(L, X), Y) = ap(insert(L, Y), X);

where
L: list; X, Y: int;
end sorted-list;

fig. 5: Specification of sorted lists
New hypothesis

\( \sigma \) infinite \( \Rightarrow \) "regularity-like" hypothesis

\( |\sigma_i| = \text{length of } \sigma_i \)

\( \forall \sigma_i \text{ s.t. } |\sigma_i| \leq k, \sigma_i(\alpha) \Rightarrow \)

\( \forall \sigma_i \in \sigma, \sigma_i(\alpha) \)

new algorithm:

let \( \Phi, \Psi \) increasing functions, \( n \) level of the test

generate via G all the instantiations \( \sigma_i \) of complexity \( \leq \Psi(n) \)

for each \( \sigma_i \), generate the instantiations of the resulting equation using algorithm 1 with input \( \Psi(n) \)

\( \Rightarrow \) Tn is acceptable

\[ G \neq \text{PROLOG} \]

\( le(x,y) = \text{true} \text{ & sorted (append (e,x)=true} \Rightarrow \in \text{insert (append (e,x),y)=true} \text{, append (append (e,x),y))} \)

? sorted (append (e,x),true), le(x,y, true).

\( \Rightarrow \)

\( L = \text{empty , } X = 0 , Y = - ; \)

\( L = \text{empty , } X = s(0) , Y = s(-) ; \)

\( L = \text{empty , } X = s(s(0)) ; Y = s(s(-)) \)

\( \ldots \)

\[ G = \text{SLOG ?} \]

logic interpreter based on narrowing

equality is handled \( \Rightarrow \) function definition

global backtracking

? sorted (append (e,x)) = true), le(x,y) = true,

le (complexity (e), n) = true,

le (complexity (x), n) = true,

le (complexity (y), n) = true.

\( \Rightarrow \)

\( e = \text{empty , } X = 0 , Y = - ; \)

\( e = \text{empty , } X = s^{-1}(0) , Y = s^{-1}(-) ; \)

\( e = \text{append (empty ,} 0) , X = 0 , Y = - ; \)
/* complexity(L) = 1 and complexity of X and Y ≤ 3 */
L = el, X = 0, Y = _;
L = el, X = succ(0), Y = succ(_);
L = el, X = succ(succ(0)), Y = succ(succ(_));

/* complexity(L) = 2 and complexity of X and Y ≤ 3 */
L = ap(el,0), X = 0, Y = _;
L = ap(el,0), X = succ(0), Y = succ(_);
L = ap(el,succ(0)), X = succ(0), Y = succ(_);
L = ap(el,0), X = succ(succ(0)), Y = succ(succ(_));
L = ap(el,succ(0)), X = succ(succ(0)), Y = succ(succ(_));
L = ap(el,succ(succ(0))), X = succ(succ(_));

/* complexity(L) = 3 and complexity of X and Y ≤ 3 */
L = ap(ap(el,0),0), X = 0, Y = _;
L = ap(ap(el,0),0), X = succ(0), Y = succ(_);
L = ap(ap(el,0),succ(0)), X = succ(0), Y = succ(_);
L = ap(ap(el,succ(0)),succ(0)), X = succ(succ(0)), Y = succ(succ(_));
L = ap(ap(el,succ(0)),succ(succ(0))), X = succ(succ(0)), Y = succ(succ(_));
L = ap(ap(el,succ(succ(0))),succ(succ(0))), X = succ(succ(0)), Y = succ(succ(_));
L = ap(ap(el,succ(succ(succ(0)))),succ(succ(succ(0))));

We are faced with two "concrete" lists:

A1 = (x1, x2, x3)  
A2 = (y1, y2, y3)

Are these equal at the abstract level?

x1 = y1  
x2 = y2  
x3 = y3

Are these concrete lists equal?

We can use a simple algorithm to compare them:

1. If the two lists are empty, they are equal.
2. If only one of the two lists is empty, they are not equal.
3. If the first elements of both lists are different, they are not equal.
4. If the first elements of both lists are the same, recursively compare the rest of the lists.

This algorithm would look something like this in a programming language:

```
function is_equal(list1, list2):
    if list1 is empty and list2 is empty:
        return True
    if list1 is empty or list2 is empty:
        return False
    if list1[0] != list2[0]:
        return False
    return is_equal(list1[1:], list2[1:])
```

This function would return True if both lists are equal and False otherwise.