CSC 7003 : Basics of Software Engineering

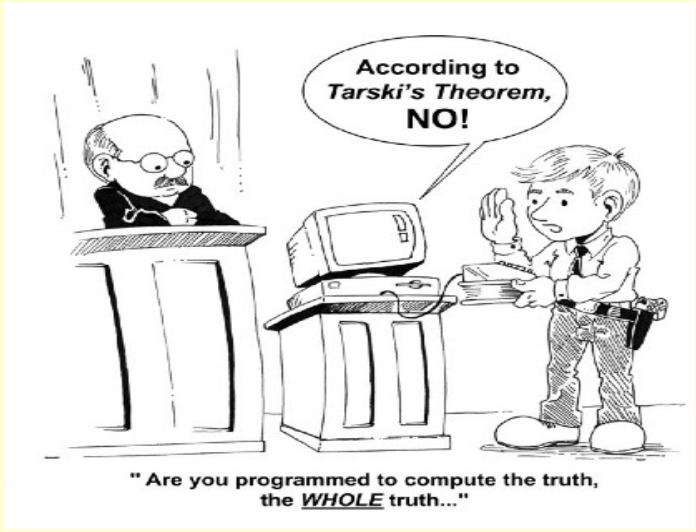
J Paul Gibson, D311

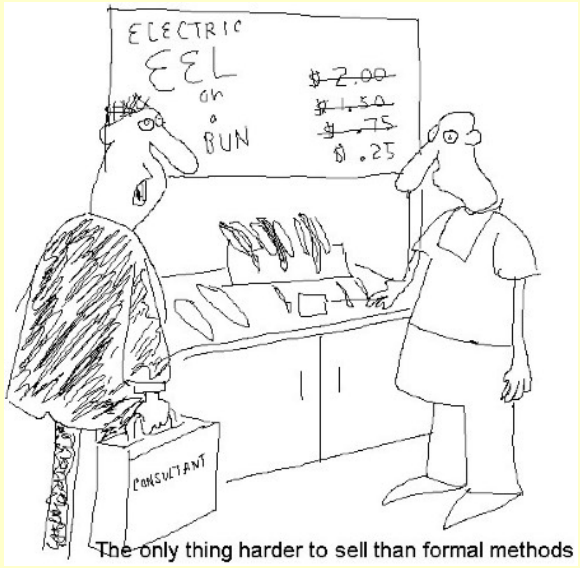
paul.gibson@telecom-sudparis.eu

http://www-public.telecom-sudparis.eu/~gibson/Teaching/CSC7003/

Rigour And Formal Methods

Rigour and Formal Methods





Rigour and Formal Methods

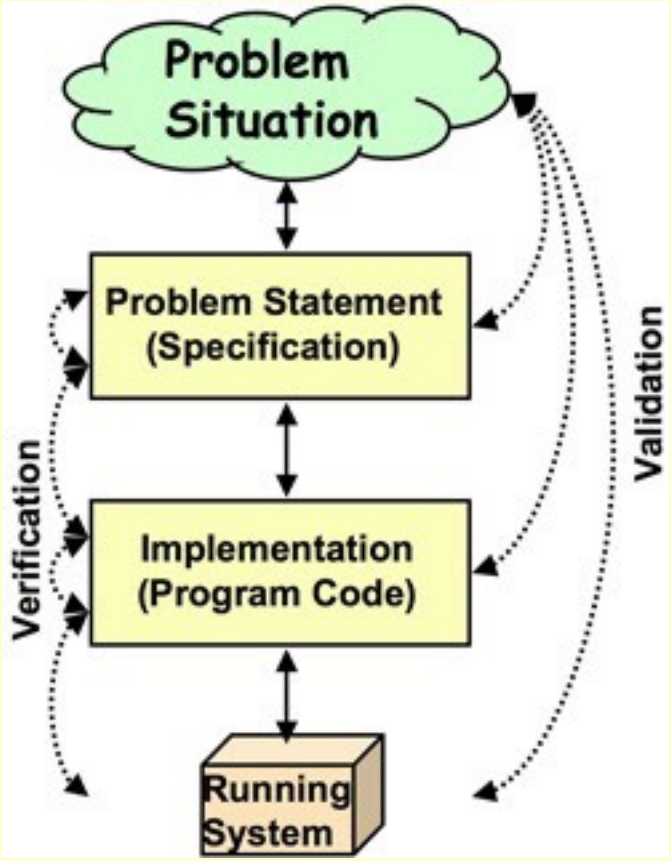
Formal methods are:

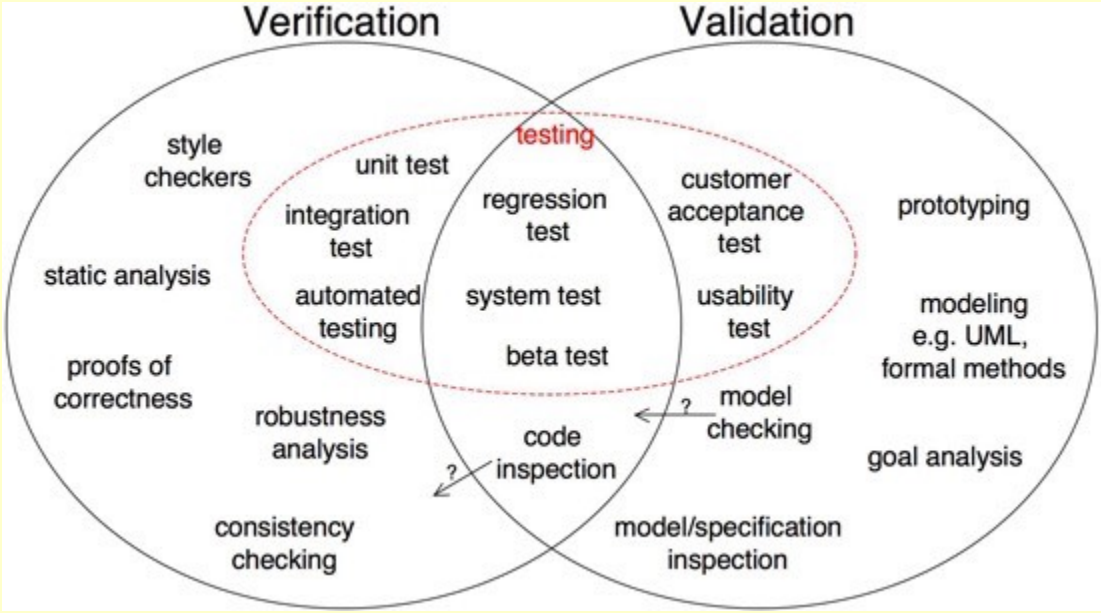
- necessary in achieving *correct* software --- fulfil its requirements
- *unambiguous* and *analysable*
- a means of improving understanding
- a modelling technique which may support non-determinism as a means of allowing implementation freedom
- facilitate re-use of analysis through *correctness preserving transformations*
- amenable to *mathematical* manipulation and reasoning
- support *rigorous testing*

But,

- they are <u>not</u> often found in industry

Verification and Validation – where are the formal methods?





http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/

Formal Methods --- technology transfer

Industrial wide acceptance is coming:

- tool support is building
- safety-critical systems are under scrutiny
- graduates are being taught the need for rigour
- engineers are realising that maths is the best tool for handling complexity
- they are being factored in as another variable in the cost-quality equation

Software engineering is about compromise --- formal methods do not need to be an all or nothing proposition so they do not remove control from the engineers they actually offer them more choice!

Correctness

Formal methods are principally concerned with maintaining correctness

Correctness is the property that a model fulfils a set of well defined requirements

In the life cycle models, formal methods can be applied *anywhere* between the initial customer-oriented requirements and the final implementation oriented design

The formal boundaries break down at either end of the process because:

- target implementation environments are not formally defined
- customer understanding of requirements is incomplete

Before we begin ...

We are not going to look at any one formal method:

- Z, VDM, ADTs, Logic, LOTOS, CSP, CCS, ML, ...

First, we are going to study simple examples which illustrate the need for formal methods (and the key concepts) …

In software process one has to identify where/when formality is appropriate.

Bottom line --- *formal methods is a state of mind, but:*

> ***a formal method is any technique concerned with the construction and/or analysis of mathematical models which aid the development of computer systems***

Typographical Re-write Systems (TRS)

A TRS is a formal system based on the ability to generate a set of strings following a simple set of syntactic rules.

Each rule is calculable --- the generation of a new string from an old string by application of a rule always terminates

A TRS may produce an infinite number of strings

TRSs can be as powerful as any computing machine (Turing equivalent)

TRSs are simple to implement (simulate) using other computational models

Using TRSs we introduce the following concepts:

proof, theorem, decision procedure, meta-analysis, structural induction, necessary and sufficient, isomorphism, meaning and consistency

Don't worry … they are very simple to understand ….

Case Study 1 --- The MUI TRS

Thanks to Hofstadter - https://en.wikipedia.org/wiki/Gödel,_Escher,_Bach

Alphabet = {M,I,U}

Strings: any sequence of characters found in the alphabet

Axiom: MI

Generation Rules: for all strings such that `x` and `y` are strings of MUI or ' ' :

- 1) `xI` can generate `xIU`
- 2) `Mx` can generate `Mxx`
- 3) `xIIIy` can generate `xUy`
- 4) `xUUy` can generate `xy`

A **theorem** of a TRS is any string which can be generated from the axioms (or any other theorem)

A **proof** of a theorem corresponds to the set of rules which have been followed to generate that theorem

Case Study 1 --- The MUI TRS (proof procedure)

Alphabet = {M,I,U}

Strings: any sequence of characters found in the alphabet

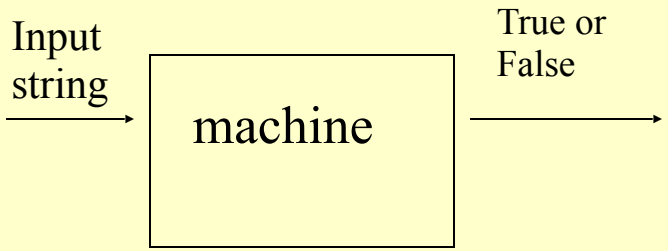
Axiom: MI

Generation Rules: for all strings such that x is a string of MUI or x =‘’ :

- 1) xI can generate xIU
- 2) Mx can generate Mxx
- 3) xIIIy can generate xUy
- 4) xUUy can generate xy

Question: can you prove the theorem MUIIU?

Question: can we automate the process of testing for theoremhood of a given string in a finite period of time?



Such a machine would be a **decision procedure** of MUI

Case Study 1 --- The MUI TRS (decision tree)

Alphabet = {M,I,U}

Strings: any sequence of characters found in the alphabet

Axiom: MI

Generation Rules: for all strings such that x is a string of MUI or x =‘’ :

- 1) xI can generate xIU
- 2) Mx can generate Mxx
- 3) xIIIy can generate xUy
- 4) xUUy can generate xy

Is this a decision procedure for the **MUI** machine? …

Construct a tree of strings, starting with the axiom at the root. Any application rule constitutes a branch of the tree. To decide if a given string is a theorem it is sufficient to keep extending the tree until the string is found.

Task: construct the top (1st 3 layers) of such a tree

Case Study 1 --- The MUI TRS (meta-reasoning)

Alphabet = {M,I,U}

Strings: any sequence of characters found in the alphabet

Axiom: MI

Generation Rules: for all strings such that x is a string of MUI or x =‘’ :

- 1) xI can generate xIU
- 2) Mx can generate Mxx
- 3) xIIIy can generate xUy
- 4) xUUy can generate xy

Question: is `IIIIUUUIIIUUUI` a theorem of the system?

Question: can you prove your answer is correct?

Note: only through *meta-reasoning* can we do this !!

Case Study 1 --- The MUI TRS (more meta-reasoning)

Alphabet = {M,I,U}

Strings: any sequence of characters found in the alphabet

Axiom: MI

Generation Rules: for all strings such that x is a string of MUI or x =‘’ :

- 1) xI can generate xIU
- 2) Mx can generate Mxx
- 3) xIIIy can generate xUy
- 4) xUUy can generate xy

The meta-property that all theorems start with an M is called a necessary but not sufficient property of theorem-hood.

Question: before we move on … is MU a theorem of **MUI**?

Now we move onto a more practical TRS ...

Case Study 2 --- The pq- TRS

Alphabet = {p,q,-}

Axiom: for any such x such that x is a possibly empty sequence of '-'s,

`xp-qx-` is an axiom

Generation Rules: for any x,y,z which are possibly empty sequences of '-'s,
if `xpyqz` is a theorem then `xpy-qz-` is a theorem

Question: is there a decision procedure for this formal system?

Hint: all re-write rules lengthen the string so …?

Case Study 2 --- The pq- TRS

Alphabet = {p,q,-}

Axiom: for any such x such that x is a possibly empty sequence of '-'s,

xp-qx- is an axiom

Generation Rules: for any x,y,z which are possibly empty sequences of '-'s,

if xpyqz is a theorem then xpy-qz- is a theorem

Why is the pq- TRS practical?

Because it provides us with a formal model of a mathematical property: the addition of integers ---

- `--p---q-----` is a theorem and "2+3=5" is true
- `--p-q--` is a non-theorem and "2+1=2" is false

Case Study 2 --- The pq- TRS interpretation

If we interpret

- p as plus
- q as equals
- and a sequence of n '-'s as the integer n

then we have

a means of checking x+y=z for all non-negative integers x,y and z

We say that pq- is **consistent** (under the given interpretation) because all theorems are true after interpretation

We say that pq- is **complete** if all true statements (in the domain of interpretation) can be generated as theorems in the system.

We say that the interpretation is **isomorphic** to the system if the system is both complete and consistent

Case Study 2 --- The pq- TRS extension

The pq- system is isomorphic to a very limited domain of interpretation (but maybe that is all that is required!)

Normally, to widen a domain we can

add an axiom

add a generating rule

For example, what happens if we add the axiom:

`xp-qx.`

Using this, we can generate many new theorems!

Question: with this new axiom what about completeness and consistency?

Answer: the new, extended system is not consistent with our interpretation.

Case Study 2 --- The extended pq- TRS reinterpreted

After extension,

`--p--q---` is now a theorem but 2+1=2 is not true

To solve this problem we can re-interpret for consistency ---

interpet q as " >= "

However, we have now lost completeness ---

"2+5 >= 4" is true (in our domain of interpretation) but

`--p-----q----` is a non-theorem

Note: this is a big problem of mathematics (c.f Church) ---

it is not possible to have a complete, decidable system of ***mathematical properties*** *which is consistent*

if all the theorems that can be checked are consistent then there are some things which we would like to be able to prove as theorems which the system is not strong enough for us to do

Case Study 3 --- A tq- TRS

Question:

- can you define a TRS for modelling the multiplication of two integers
- can you show that it is complete and consistent

Interpretation:

- t as times
- q as equals
- sequences of '-'s as integers

BACK TO THE SOFTWARE PROCESS …

PBL - Imagine you were asked to implement a function, f say, to calculate the *ith* prime number.

Thus, given the primes to be 2,3,5,7,11,13,17,19,…,

f(1) =2, f(2) = 3, f(3) =5, …

I assume you could all code this directly in C, C++, Python, Java, Prolog …
How many of you could prove your code was correct?
Where would you even start?

First: formalise requirements
Second: transform requirements into design and prove transformation to be correct
Third: keep correctly transforming design until it is directly implementable
Fourth: implement it

Question - but is this process/sequence fully formal?

A TRS for formally defining if a number is prime

Note: easier to do in other formal languages/methods because the necessary concepts (like integers and lists are part of the language)

But, with the TRS we define just what we need and use it only where needed.

In software process it is this targeting (with the minimum force necessary) which is best …

Question: can you write a TRS for deciding if a given number is prime?

Hint: if not, try to break the problem down into bits

In a more realistic approach, we introduce Abstract Data Types…. IMHO the most powerful and universally applicable software process formal methods tool.

From TRSs to Abstract Data Types (ADTs)

ADTs are a very powerful specification technique which exist in many forms (languages).

These languages are often given operational semantics in a way similar to TRSs (in fact, they are *pretty much equivalent*)

Most ADTs have the following parts ---

- A type which is made up from sorts
- Sorts which are made up of equivalent sets
- Equivalent sets which are made up of expressions

For example, the integer type could be made up of

- sorts integer and boolean
- 1 equivalence set of the integer sort could be {3, 1+2, 2+1, 1+1+1}
- 1 equivalence set of the boolean sort could be {3=3, 1=1, not(false)}

Case Study 4: A simple ADT specification

TYPE integer SORTS integer, boolean

OPNS

0:-> integer

succ: integer -> integer

eq: integer, integer -> boolean

+: integer, integer -> integer

EQNS forall x,y: integer

0 eq 0 = true; succ(x) eq succ(y) = x eq y;

0 eq succ(x) = false; succ(x) eq 0 = false;

0 + x = x; succ(x) + y = x + (succ(y));

ENDTYPE

Case Study 4: A simple ADT specification

TYPE integer SORTS integer, boolean

OPNS

0:-> integer

succ: integer -> integer

eq: integer, integer -> boolean

+: integer, integer -> integer

EQNS forall x,y: integer

0 eq 0 = true; succ(x) eq succ(y) = x eq y;

0 eq succ(x) = false; succ(x) eq 0 = false;

0 + x = x; succ(x) + y = x + (succ(y));

ENDTYPE

Question: how do we show, for example ---

- 1+2 = 3,
- 3+2 = 4+1,
- 2+2 != 3+2

Case Study 4: A simple ADT specification

TYPE integer SORTS integer, boolean

OPNS

0:-> integer

succ: integer -> integer

eq: integer, integer -> boolean

+: integer, integer -> integer

EQNS forall x,y: integer

0 eq 0 = true; succ(x) eq succ(y) = x eq y;

0 eq succ(x) = false; succ(x) eq 0 = false;

0 + x = x; succ(x) + y = x + (succ(y));

ENDTYPE

Note: this model is complete and consistent with respect to the modelling of the addition of integers (like the TRS pq-)

Question: extend this model to include multiplication

Case Study 4: An equivalent ADT specification

Consider changing the original specification to make explicit the fact that **x+y = y +x**, for all integer values of x and y:

TYPE integer SORTS integer, boolean

OPNS

0:-> integer

succ: integer -> integer

eq: integer, integer -> boolean

+: integer, integer -> integer

EQNS forall x,y: integer

0 eq 0 = true; succ(x) eq succ(y) = x eq y;

0 eq succ(x) = false; succ(x) eq 0 = false;

0 + x = x; succ(x) + y = x + (succ(y));

x+y = y+x;

ENDTYPE

Note: this does not change the meaning of the specification but it may affect the implementation of the evaluation of expressions

Case Study 4: Evaluation termination

If expressions are evaluated as left to right re-writes (as they often are) then evaluation may not terminate:

3 +4 = 4+3 may be re-written as

4+3 = 3+4 which may be re-written as

3+4 = 4+3 …

Consequently, there are 3 important properties of ADT specifications:

- completeness
- consistency
- evaluation termination/convergence

Case Study 4: Incompleteness, inconsistency and termination

Not having enough equations can make a specification incomplete. For example, the integer ADT specification would be incomplete without the equation:

$$0 \text{ eq } 0 = \text{true}$$

Having too many equations can make a specification inconsistent. For example, the integer ADT specification is inconsistent if we add the equation:

$$x + \text{succ}(0) = x$$

but adding the equation:

$$x + \text{succ}(0) = \text{succ}(x)$$

would not introduce inconsistency (just redundancy)

Changing the equations may affect termination:

$$0 + x = x \text{ to } x + 0 = x$$

would introduce non-termination to the original ADT specification

Case Study 5 --- A Set ADT specification

```
TYPE Set SORTS Int, Bool
OPNS
empty:-> Set
str: Set, int -> Set
add: Set, int -> Set
contains: Set, int -> Bool
EQNS forall s:Set, x:Int
contains(empty, int) = false;
x eq y => contains(str(s,x), y) = true;
not (x eq y) => contains(str(s,x), y) =
                        contains(s,y);
contains(s,x) => add(s,x) = s;
not(contains(s,x)) => add(s,x) = str(s,x)
ENDTYPE
```

Notes:

- use of str and add
- preconditions
- completeness?
- consistency?

Question:

add operations for --

- remove
- union
- equality

Case Study 6: Set verification

We would like to verify the following properties:

- $e \notin (S\text{-}e) = \text{true}$
- $e \in S1 \cup S2 \Rightarrow e \in S1 \ \vee \ e \in S2$

Proof technique: structural induction on the ADT specification

Question: try it yourselves to see how it goes ...

Invariant Property: verify also that a set never contains any repeated elements

Back to the Primes Proof

Question:

- write an ADT specification of a list of integers
- include a means of verifying that it is ordered
- include a function for returning the length

All that is left to do is plug the two parts together and we have a formal specification (and implementation) of our prime problem requirements.

Question:

- what use is it to us?

Further Reading

Hall, Anthony. "Seven myths of formal methods." *Software, IEEE* 7.5 (1990): 11-19.

Bowen, Jonathan P., and **Michael G. Hinchey**. "Seven more myths of formal methods." *IEEE software* 12.4 (1995): 34-41.

Clarke, Edmund M., and **Jeannette M. Wing**. "Formal methods: State of the art and future directions." *ACM Computing Surveys (CSUR)* 28.4 (1996): 626-643

Woodcock, Jim, et al. "Formal methods: Practice and experience." *ACM Computing Surveys (CSUR)* 41.4 (2009): 19.

Clarke, Edmund M., **E. Allen Emerson**, and **Joseph Sifakis**. "Model checking: algorithmic verification and debugging." *Communications of the ACM* 52.11 (2009): 74-84.

Bowen, Jonathan P., and **Mike Hinchey**. "Ten Commandments of Formal Methods… Ten Years On." *Conquering Complexity*. Springer London, 2012. 237-251.