UPPSALA UNIVERSITET

# About Formal Methods in Software Development

Lars-Henrik Eriksson

lhe@it.uu.se,

http://www.it.uu.se/katalog/lhe

# The software development process

- Requirements capture
- Specification

  Traditionally done using plain language, diagrams, tables ...
- Validation (are we building the right system?)

  Traditionally done by inspection, prototyping ...
- Design

  Specify the architecture and data structures of the software
- Implementation

  Programs written in a programming language
- Verification (are we building the system right?)

  Traditionally done by testing
- Debugging

  Try to find out where the program goes wrong

# Mathematical models

Other engineering disciplines would never accept a methodology that relies so heavily on testing.

Instead, designs are analysed using *mathematical models* from e.g.

- material science (e.g. bridge construction)
- control theory (e.g. process control)
- electricity theory (e.g. electronics)

… giving a much lower probability of incorrect design.

*Formal methods* is a methodology with a similar role in the development of software and related systems.

# Formal methods

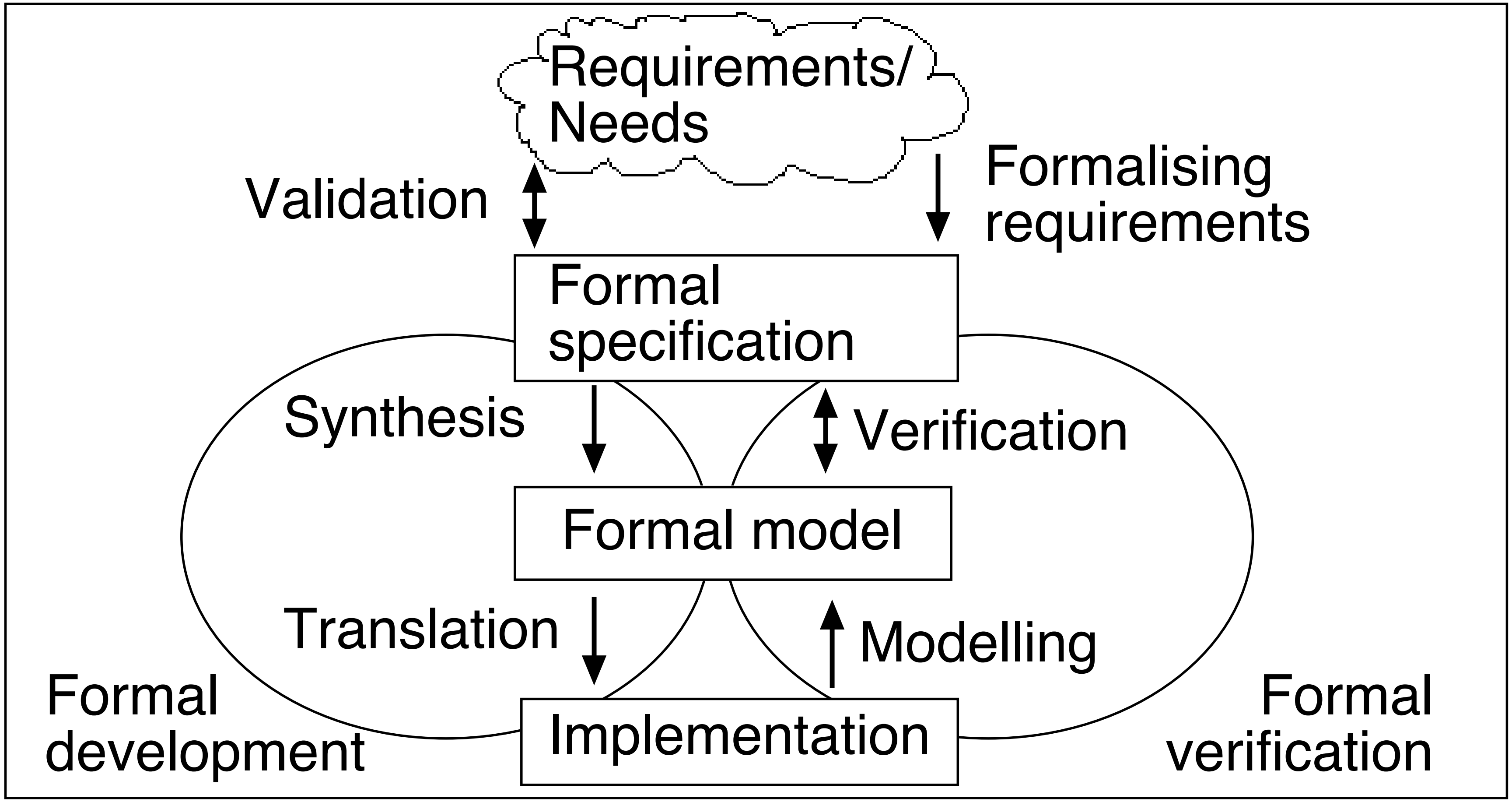
…are based on *formal*, that is

*mathematical/logical*

- requirements specifications
- design models
- *proofs* of relations between design models and specifications.

*Discrete mathematics* and *mathematical logic* is used.

# Concepts of formal methods



Requirements/ Needs

Validation

Formalising requirements

Formal specification

Synthesis

Verification

Formal model

Translation

Modelling

Formal development

Implementation

Formal verification

# Formal specifications

The formal specification should give an exact description of the requirements of the system. Requirements can concern e.g.:

- Functionality
- Time aspects
- Fault tolerance

Incomplete requirements can still be of good use.

Requirements can als be code-oriented, e.g. ”null pointers must not be dereferenced”.

# Sample specification

Take the problem of sorting a collection of objects:

$$\forall x,y.(\mathrm{sort}(x,y) \Leftrightarrow \mathrm{sorted}(y) \wedge \mathrm{permutation}(x,y))$$

Or, in natural language:

For all collections, x and y, y is the result of sorting x if and only if

- y is a sorted collection
- y is a rearrangement (permutation) of x

The concepts ”sorted” and ”permutation” then have to be defined, in turn.

# Validating the specification

How do you know that the formal specification correctly represents the necessary requirements?

A good understanding of the formal language used is needed in order to avoid mistakes, but the specification must also be *validated*.

A formal specification can be validated with the aid of a computer using software tools to do:

- Animation (the computer simulates the specified system)
- Testing (also by *proving* properties of the specification)

# Using formal specifications

A formal specification is a prerequisite for using formal methods, e.g. for the verification of programs and systems.

The specifications are of value even if formal methods are not otherwise used. Among other things, they can be used to

- Improve understanding of requirements.
- Improve communication of requirements.
- Automatically construct test data.

A large part of software errors are caused by poor and/or incorrect requirements specifications.

# Formal domain theories

Using formal methods notation, mathematical theories of an application domain can be made.

E.g. in the ”railway domain” a domain theory describes how tracks are laid out, how trains move etc.

Such theories define precise concepts and notions that can be used better understand the domain and to express requirements.

# Formal verification

Formally verifying the system means proving that the model of the implementation fulfils the requirements as expressed by the formal specification.

Since proofs are exact, a successful proof ensures that the requirements are satisfied (provided that the proof has been carried out correctly).

Proofs can be carried out

- Automatically by the computer (best, but not always possible)
- With computer support

(In the old days also by hand, but that is not practical except for toy examples.)

# Formal development

…means that the implementation is constructed formally from the specification.

Typically the end result is in a formal notation that can be automatically translated into ”ordinary” code (C, Ada etc.)

If every step in the construction process is correct, the result is guaranteed to fulfil the requirements.

Formal development is a more difficult problem than verification, since there is not a single correct solution.

Fully automated development is not possible in practise (but you can get close… se Siemens experience further on)

# Java Modelling Language

- A ”lightweight” formal methods notation.
- Simple code-oriented specifications.
- Annotations are added to a Java program to express requirements on methods.
- Supported by various software tools such as ESC/Java2 which prove that the Java program satisfies the requirements.
- Microsoft’s Spec# is a similar approach for C# programs.

# JML example

```
public class BankingExample {
 public static final int MAX_BALANCE = 1000;
 private int balance;
 private boolean isLocked = false;

 //@ invariant balance >= 0 &&
              balance <= MAX_BALANCE;

 //@ requires amount > 0;
 //@ ensures balance == \old(balance) + amount;
 //@ assignable balance;
 public void credit(int amount) { ... }

 //@ ensures isLocked == true;
 public void lockAccount() { ... }

... }
```

# SLAM

- A program analysis tool developed by Microsoft
- Addresses the problem of instability of the Windows operating system caused by buggy third-party device drivers.
- Automatically makes formal verification of C programs.
- ”Turn-key” tool with built-in requirements.
- Checks that the program to be verified correctly uses Windows Kernel API calls related to device drivers and that memory management is  done correctly.

# The B-method

The *B-method* is a *formal method* used for

- Formal specification of software (using the *Abstract Machine Notation – AMN*)
- Writing executable programs (using the *B0* subset of AMN)
- Proving consistency of specifications and correctness of programs

Characteristics:

- *Model-based specification*
- *Refinement*

The B-method is supported by *software tools* such as

- Atelier B
- B-Toolkit
- ProB

# The role of B in software development

- Specification

  *Wholly or in part written in AMN.*

- Validation (are we building the right system?)

  *Proving correctness theorems, animating the specification...*

- Design

  *Design specifications wholly or in part written in AMN.*

- Implementation

  *Programs written in the B0 subset of the AMN.*

  *Automatic translation to C or ADA.*

- Verification (are we building the system right?)

  *Refinement proof. Testing should not be needed.*

- Debugging

  *You don’t need this (at least not in the traditional sense)*

# About the B-method

The B method was developed with practical software development in mind. It brings together ideas from various areas of computer science (and mathematics). Some of them are:

- Axiomatic set theory (Zermelo-Fraenkel)
- Model-based specifications (Z, VDM-SL)
- Pre- and postconditions
- Design by contract
- Invariants
- Guarded commands
- Weakest precondition semantics
- Hoare logic (axiomatic semantics)
- Refinement calculus
- Stepwise refinement

# B notation example

```
MACHINE Exp
OPERATIONS rr <-- exp(bb,ee) =
  PRE bb:NAT & ee:NAT & bb**ee:NAT
  THEN rr := bb**ee END
END

IMPLEMENTATION ExpI
REFINES Exp
OPERATIONS pp <-- exp(bb,ee) =
  VAR kk IN
    pp := 1; kk := ee;
    WHILE kk>0 DO
      pp := pp*bb; kk := kk-1
      INVARIANT pp=bb**(ee-kk) & ee>=kk & kk:NAT
      VARIANT kk
    END
  END
END
```

# A stack specification in B

```
MACHINE Stack
CONSTANTS maxsize
SETS ELEMENTS
PROPERTIES maxsize:NAT
VARIABLES stack
INVARIANT stack:seq(ELEMENTS)&size(stack)<=maxsize
INITIALISATION stack := <>
OPERATIONS
   xx <-- get = PRE stack /= <>
                THEN xx := last(stack) END;
   push(xx)   = PRE xx:ELEMENTS &
                    size(stack)<maxsize
                THEN stack := stack<-xx END;
   pop        = PRE stack /= <>
                THEN stack := front(stack) END
END
```

# A stack implementation in B

```
IMPLEMENTATION StackI
REFINES Stack
VALUES ELEMENTS = INT; maxsize = 100
CONCRETE_VARIABLES array, currentsize
INVARIANT array:(1..maxsize)-->ELEMENTS &
           currentsize:0..maxsize &
           !ii.(ii:1..currentsize =>
                stack(ii) = array(ii)) &
           currentsize = size(stack)
INITIALISATION array := (1..maxsize)*{0};
               currentsize := 0
OPERATIONS
   xx <-- get = xx := array(currentsize);
   push(xx)   = BEGIN
                  currentsize := currentsize+1;
                  array(currentsize) := xx
                END;
   pop        = currentsize := currentsize-1
END
```

# Siemens Transportation Systems

## (formely MATRA transport)

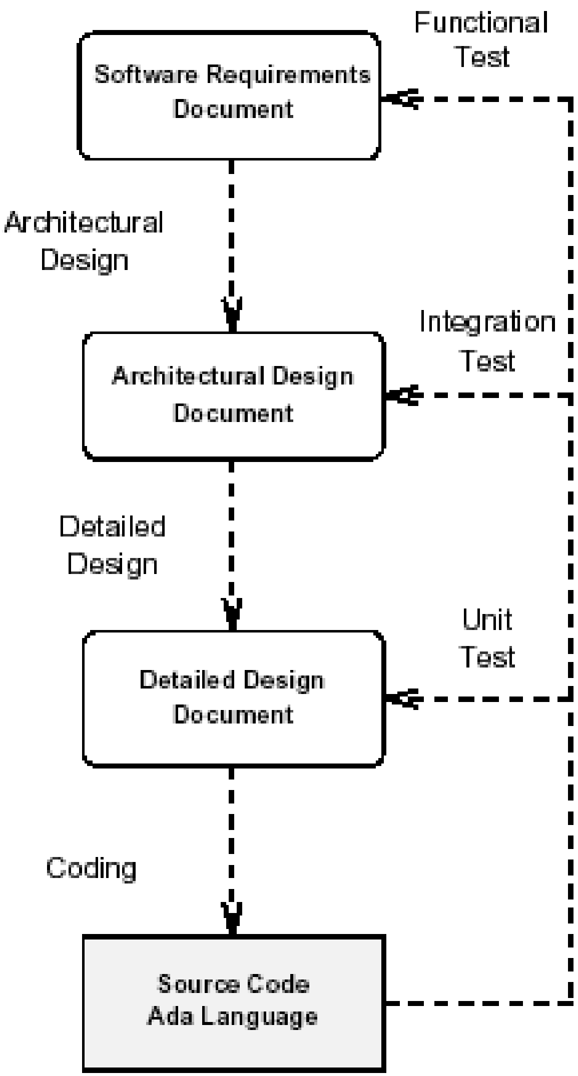
## use of B

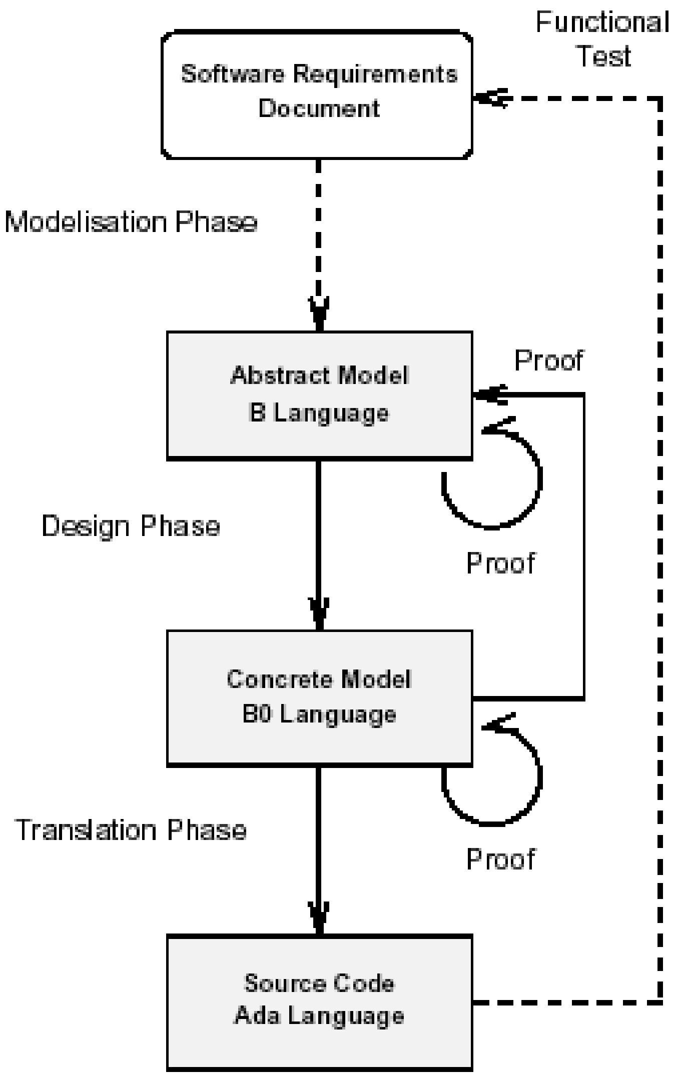
Develops ATP/ATO (Automatic Train Protection/Operation) systems.

Have used the B-method for some 15 years.

Complete formal development process from specifications to executable code.

Safety-critical part of Météor (software for Paris metro line 14) comprises 86 000 lines of Ada code, all developed formally.


Software Requirements Document
Functional Test
Architectural Design
Architectural Design Document
Integration Test
Detailed Design
Detailed Design Document
Unit Test
Coding
Source Code Ada Language


Software Requirements Document
Functional Test
Modelisation Phase
Abstract Model B Language
Proof
Design Phase
Proof
Concrete Model B0 Language
Translation Phase
Proof
Source Code Ada Language

# Siemens Transportation Systems some statistics

| | | |
|---|---|---|
| Project: | Météor (1996)<br>(Paris metro 14) | CBTC (2004)<br>(New York<br>Canarsie Line) |
| Staff: | 10 | 4 |
| Proof obligiations | 23 000 | 41 000 |
| Automatically proved | 75% | 86% |
| Effort, man-years | 17 | 4 |
| Lines of generated code | 86 000 | > 100 000 |
| Reliability | ”Still runs v1.0”<br>(in 2009) | |

# Siemens Transportation Systems experiences

≈90% (today) of all proof obligations proved automatically.

Several errors detected during development by failed proof.

Resulting code is virtually error–free.

No testing done until the system test.

Cost of developing safety-critical software approaching that of non safety-critical software.

(Important obstacle to developing non safety-critical software with B is lack of qualified developers.)

# Some other examples

- Airbus A340/A380 flight control software. Automatic code generation from formal design models using the SCADE tool.
- Mondex smart card (electronic purse). Design (including security model) formally modelled in Z and verified. (Mondex has become a ”benchmark problem” for formal methods.)
- Tokeneer Secure Entry System. An access control system used by the US NSA utilising biometric tests. Formal models in Z and partially formally verified code developed in SPARK ADA. The system has been made publically available as an example of a major development of high-integrity software.

# How are formal methods used in industry?

- Some – steadily increasing
- Greatest acceptance with safety-critical applications and digital circuit design (remember the Pentium I floating-point division error in the 1990s).
- Several process and quality standards demand or recommend the use of formal methods (e.g. UK DEF STAN 00-55, Cenelec EN5012x series)
- Automatic FM-based analysis tools such as Microsoft’s SLAM are finding increasing use.

# Results from a survey in 2009

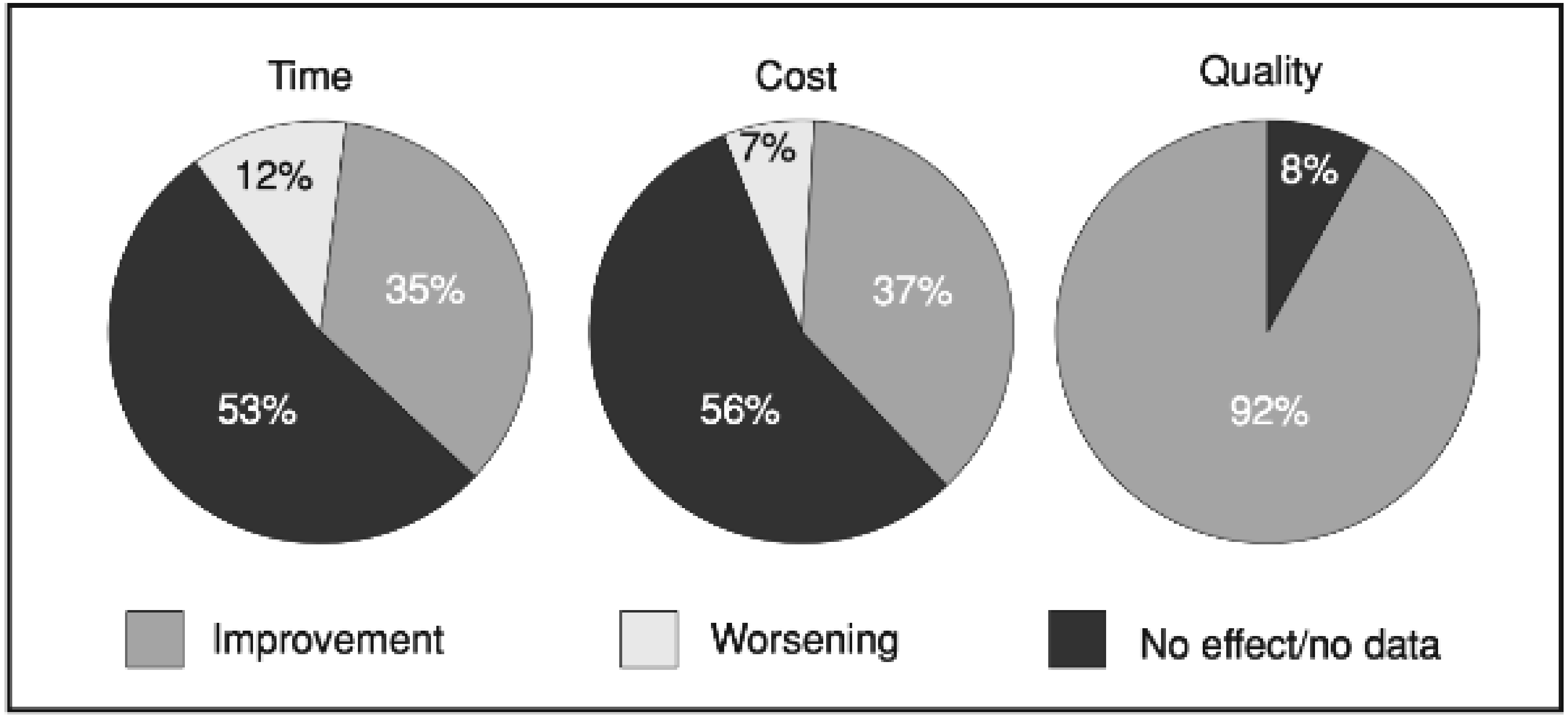


**Fig. 6.** Did the use of formal techniques have an effect on time, cost, and quality?

[Woodcock et. al., *Formal Methods: Practice and Experience ACM Computing Surveys* no. 4, vol. 41 (October 2009)].

# A controlled experiment

Few studies done on how FM affect software development projects.
Controlled experiments difficult: high cost and risk.

British Aerospace Systems and Equipment Ltd (BASE) study in the 1990s: Develop a simple (but realistic!) security-critical program *twice* – with and without formal specifications being used.

The specification notation used was VDM-SL which is similar in concept to the B AMN notation.

Traditional development process using the ”V” model.
Input – a natural language customer requirements document.
Output – software written in C + various development metrics.

Same resources to both efforts. Similar staff qualifications.

# General results

Both teams delivered within schedule and budget.

Engineers had little trouble learning and using VDM-SL.
Tool support essential!

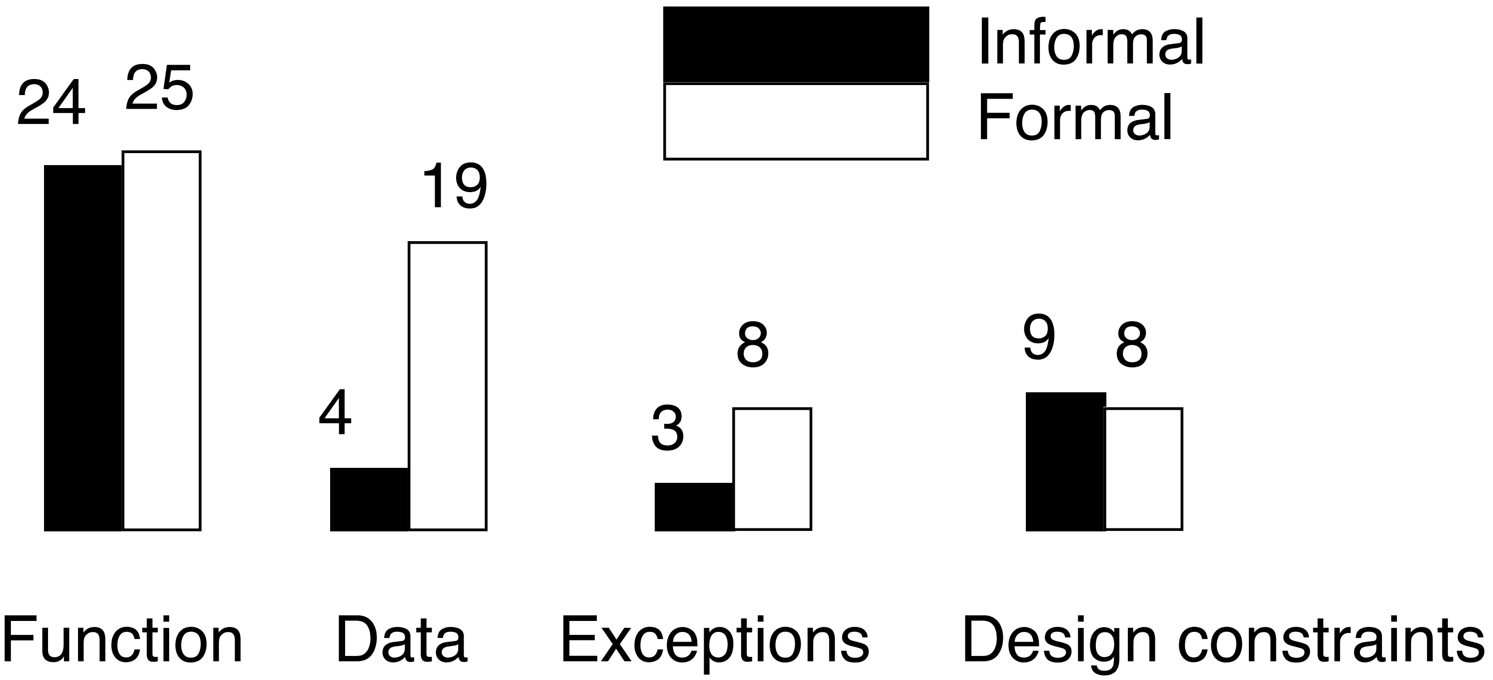
System analysis took longer in the formal project.
Design and implementation took less time.

Informal development produced slow, poor-quality software because of a mistake detected late in the project.
Strict quality standards would force development to start over!

No certain conclusions can be drawn, but this all concurs with ”accepted wisdom” in the FM community.

# Queries about customer req’s

Queries were logged and analysed. Formal team made 50% as many queries as informal team.



# Formal Methods do not do everything

- The whole of the development process is not covered by formal methods.
- Not all kinds of questions can be handled with formal methods (in practise)

Testing is still required – but to a lesser extent.

# Formal Methods courses at the IT dept.

- Provably Correct Software (B-method)
- Verification methods (distributed systems, communication protocols)