Designing reliable systems using formal methods:

Models & languages for model checking

ENSIMAG 3e année, spécialité ISI

&

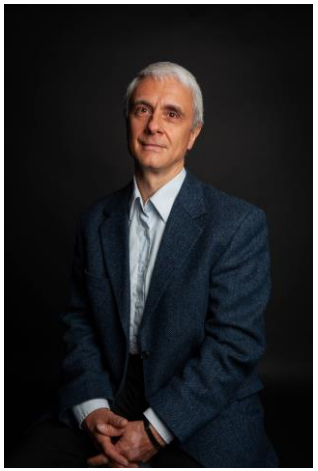
MOSIG Master 2

(1st part of: « System Design: Concurrency, Real-Time, Stochastics and Analog/Digital »)

Lecturers

Frédéric LANG,
Chargé de recherche Inria
Frederic.Lang@inria.fr
04.76.61.55.11

Radu Mateescu,
Directeur de recherche Inria
Radu.Mateescu@inria.fr
04.76.61.54.86



Inria/Convecs team, Montbonnot

Course overview (1/2)

1. Introduction (Tue 5/10) - FL
   - Contribution and impact of formal methods in development processes
   - Foundations of concurrency and real-time
2. Communicating automata (Tue 12/10) - FL
3. Timed automata (Tue 19/10) - FL
4. Lab session (Tue 26/10) - FL
5. Process algebras (Thu 25/11) - FL

Course overview (2/2)

6-7. LNT (Tue 30/11 & 7/12) - FL

8. Lab session (Tue 14/12) - FL

9-10. Temporal logic (Tue 4/01 & 11/01) - RM

11. Lab session (Tue 18/01) - RM

12. Revision (Tue 25/01) - RM

Exam: probably 1st week of Feb. 2022

Organisation

- Chamilo :

  http://chamilo.grenoble-inp.fr/courses/ENSIMAG5MMMVSC7

  - Teachers contacts
  - Course material
- Evaluation
  - An examination mark (documents authorised)
- Lab sessions are not graded, but necessary
- You may send questions by e-mail

Improving development processes using formal methods

What is a FM (Formal Method)?

- Method to help developing reliable (software & hardware) systems
- Based on **formal models**:
  - **Abstract** and unambiguous system descriptions
  - In languages with mathematically-defined semantics
- Advantages:
  - Reference: no interpretation divergences, no disputes
  - Allows (some aspects of) system correctness to be formally verified
  - Can be used at several steps of system engineering: from design to implementation & test

Numerous FM

- Set-theoretic methods (sequential)
  VDM, Z, B / Event-B, …
- Algebraic specifications (sequential)
  Larch, ASM, OBJ, …
- Petri nets (concurrent)
- Process algebras (concurrent)
  CCS, CSP, LOTOS / LNT
- …

Some knowledge bases

- Wikipedia and Formal Methods Wiki:
  many formal methods are enumerated!
- http://www.fmeurope.org: some references
- http://www.cs.indiana.edu/formal-methods-education/Tools: list of software tools
- Paper on FM published in Science et vie (in French, February 2011)
  In Chamilo

Why so many FM?

- Same situation as for programming languages
- For long, it has been more a logic of offer (by academia) than a logic of demand (by industry)
- Each FM targets a precise domain: description of architecture, data, sequential processing, concurrency, real-time, etc.
- Each FM has its strengths and weaknesses

« Executable » FM

- A FM is executable if executable code can be generated automatically from the formal model
- Some FM (VDM, Z, temporal logic, etc.) are not executable: they describe what a system must do, but not how it must do it (« declarative » approach), not even whether it can be done
- In this course, we will see both types of FM (executable or not), which can be combined

Classification wrt. two criteria

taxonomy	Non formal language	Formal language
Non executable language	UML	VDM, Z, temporal logic, …
Executable language	Many programming languages, …	Functional languages (ML), Lustre, LNT, …

Cost and profit of FM

The initial cost of FM

FM have the same disadvantages as any quality improvement effort:

- The initial cost of formal modeling is more important than traditional approaches (expertise)
- The effort put in the formal modeling does not always immediately improve the final product delivered to the client

Right, but they also have advantages…

1. Better quality of specifications

- With FM, more care is put in the early phases of the project
- Much better specifications are obtained, which will serve as reference documentation for the project
- This way, long term maintenance will be easier

2. Simplification of the coding phase

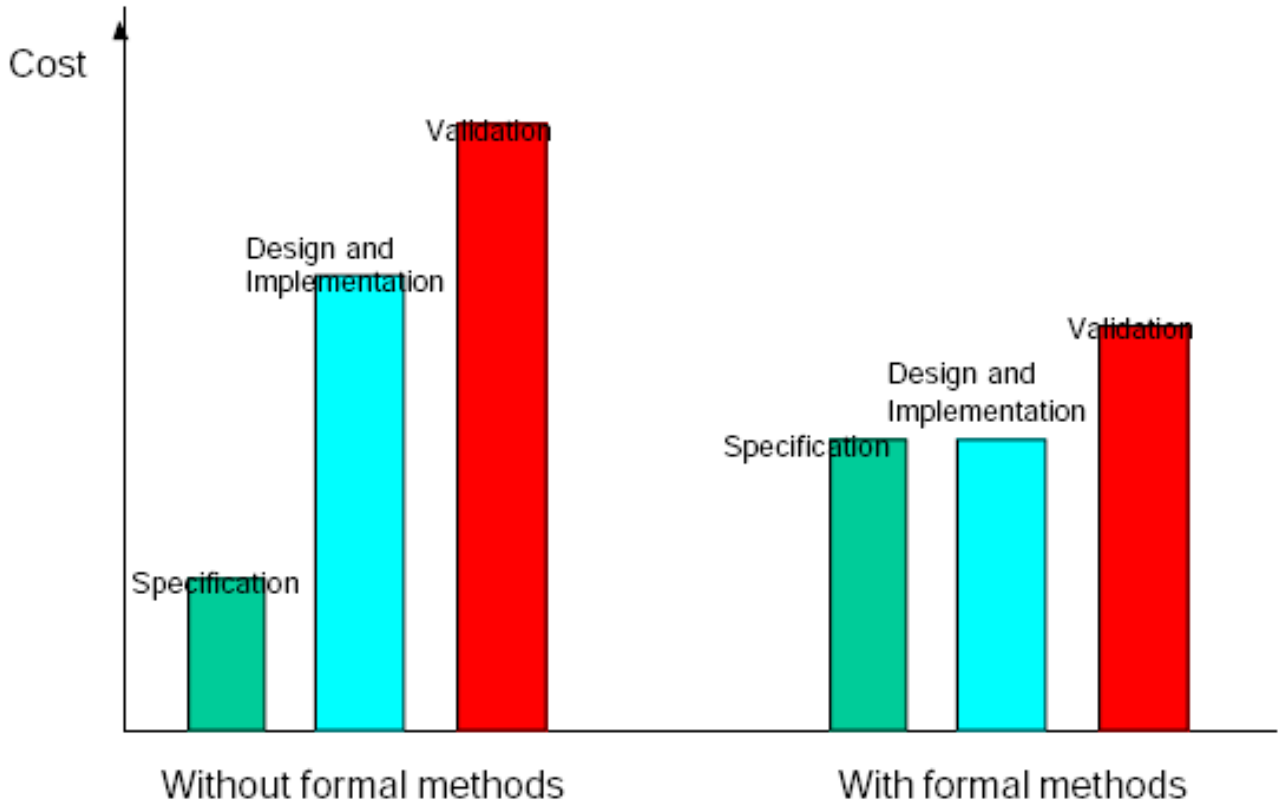
- With FM, the coding phase is simpler, because the programmer knows precisely what (s)he must do

- Ambiguities are eliminated: programmers cannot anymore make mistakes by incorrectly interpreting the contents of the specifications they must implement

3. Earliest error detection

- The later an error is detected in the product lifecycle, the more expensive its correction
- The worst errors are those detected when the product has already been delivered
- With FM, errors are detected earlier, during formal modeling and verification

New distribution of effort and cost



Detecting errors earlier reduces the cost and duration of tests

Additional outcome

Formal specifications have the following two additional outcomes:

- Automated verification:

  allows to detect automatically the errors that are present in the formal model, which increases the number of errors discovered early

- Code and test generation:
  allows to automate the coding and testing phases

Automated verification (1/4)

- Idea: use the computing power to analyse the formal model and:
  - Either prove that the model is correct
  - Or detect errors automatically

- It works for larger and larger examples

- As for chess player programs, "brute force" (exploration of all possible cases) and "heuristics" (smart strategies to limit the number of cases) are combined

Automated verification (2/4)

What do we verify?

- Functional (or qualitative) aspects
  - Absence of deadlock
  - Determinism
  - Absence of unspecified receptions, etc.
- Non-functional (or quantitative) aspects
  - Response time
  - Performance
  - Memory consumption
  - Electric consumption, etc.

Automated verification (3/4)

Two main approaches for functional verification:

- Proof (or deductive verification, theorem proving):
  a computer is used to demonstrate mathematically the result by application of logic rules

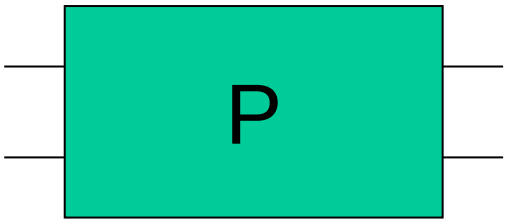
- Enumerative verification (brute force):
  a computer (or a cluster of computers) is used to enumerate and verify all possible cases

Both approaches are not mutually exclusive:
they should be combinable

Automated verification (4/4)

Two methods to verify a program P :

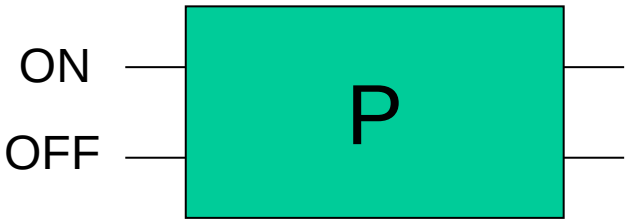
- Model checking ("*two language approach*") :
  we determine whether P satisfies a set of logic formulas φ_1, φ_2 , … φ_n that express « good functioning » properties



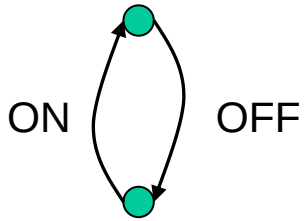
$$\begin{cases} P \models \varphi_1 \\ P \models \varphi_2 \\ \dots \\ P \models \varphi_n \end{cases}$$

|= : the *program* P *satisfies the formula* φ

- Equivalence checking ("*single language approach*") :
  we determine whether P is equivalent to (i.e., does the same thing as) another program P' that is known correct



(hide all but ON, OFF in P) ~



Additional outcome

Formal specifications have the following two additional outcomes:

- Automated verification:
  allows to detect automatically the errors that are present in the formal model, which increases the number of errors discovered early

- Code and test generation:
  allows to automate the coding and testing phases

Rapid prototyping

- If the specification is described with an « executable » FM, it can be considered as a program written in a very high-level language
- Some executable formal methods are equiped with compilers (which generate C code, for instance)
- They can be used to fastly generate prototypes that will be shown to the client
- Possibly, the coding itself can be automatized (beware the time and memory cost!)

Automated test generation

- If the formal model is executable, it can be used to generate tests automatically

- This approach reduces the testing effort

Examples of such tools:

- TGV (*Test Generation based on Verification*)
  Web: look for « TGV test » in Google
- GATeL (developed at CEA/LIST)
  Web: look for « GATEL test list » in Google

Co-simulation (intensive testing)

- Use the code produced from an executable formal model to pilot the real system
- The formal model receives the real system's outputs and sends its inputs
- Observers are used to detect any behavioural difference between the model and the real system

Impact of FM

A slow and difficult dissemination

- FM are not widespread in the software industry.
- As any (r)evolution, FM have been the object of resistance and controversy.
- FM is a young subject (~50 years) as compared to the theoretical and practical complexity of the addressed problems.
- The initial objectives were too ambitious, which has led to disapointment and distrust.
- But things have been (slowly but safely) progressing...

Difficulties related to FM (1/2)

- They require competencies in logics and mathematics that not all developers have.
- They require additional learning and reflexion effort.
- They are not a general method: they do not necessarily apply to every aspect of systems, but only to the most complex parts.

Difficulties related to FM (2/2)

- Other more simple techniques (testing, simulation, ...) have allowed software quality improvements by detecting more rapidly the errors that are « easy to find », which has reduced the interest for FM. The « difficult » errors remain…

- In the current economic competition, reactivity and « *time to market* » are more important than quality (« *low error count* »). FM reduce the number of errors,but it is not easy to predict whether they increase or reduce the development time.

Anyway

- FM have more and more technical successes.
- They are spreading progressively in the most innovative companies.
- Concerns on quality (reliability, security, etc.) start to be the object of standards.

Successes in the « hardware » domain

- FM are now commonly used for circuit and architecture designs
- There are verification teams next to development teams. For complex circuits, 70% of the effort is put on verification and test
- Example : the PSL standard (*Property Specification Language*) of the Accellera consortium
  Web : http://www.accellera.org (-> PSL)
- Designers (Intel, AMD, IBM, etc.) use formal verification tools. The biggest (IBM, Intel) even have their own labs, which develop formal verification tools

Successes in the « software » domain

- Example 1 : The SPIN model checker (Bell Labs)
  http://spinroot.com/spin/whatispin.html#X
  - The Rotterdam flood control barriers
  - The Lucent Pathstar switch
  - NASA missions: Cassini, Mars, etc.

- Example 2 : The CADP verification tools (Inria)
  http://cadp.inria.fr/case-studies

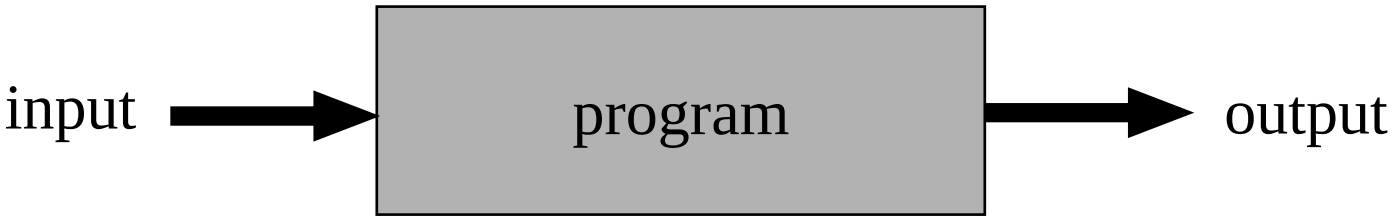
    > 200 case studies in various domains

Summary

- For specification and design, FM improve considerably the usual practice (natural language + diagrams).
- They require a sharp experience and thus concern prioritarily the « critical » systems: avionics, nuclear plants, transport, circuit design, security, etc.
- Their implementation cost can be compensated by an outcome on some development steps (automatic coding, validation, test, etc.). They can thus deeply modify the traditional development cycles.
- The FM must be chosen according to the nature of the problem: sequential, concurrent synchronous, concurrent asynchronous, real-time, etc.

Concurrency and real-time

Transformational programs



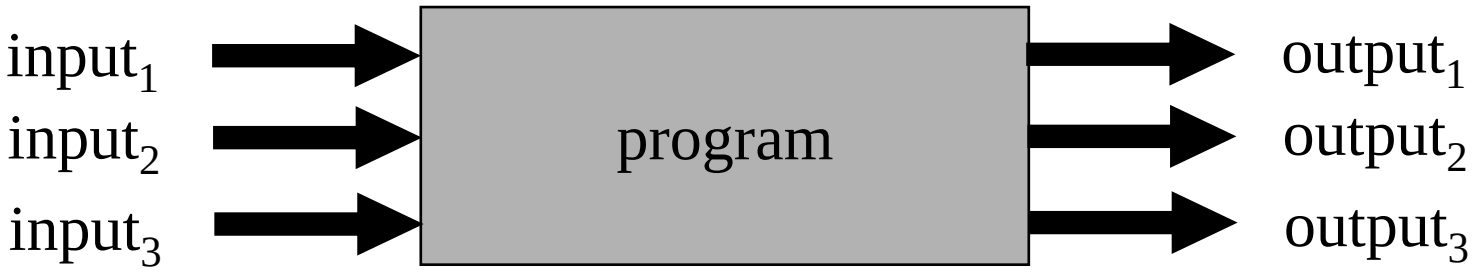
Caracteristics :

- Sequential behaviour
- Termination is normal
- Output is a function of the input: output = f(input)

Examples :

- Algorithmic programs (C, C++, Ada, …)
- Functional programs (ML, Scheme)

Reactive programs (1/3)



Caracteristics :

- Cyclic behaviour
- Termination is anormal
- Receive inputs and respond by outputs

Examples :

- Operating systems, Unix daemons (login)
- Graphical interfaces

Reactive programs (2/3)

- A same input can produce different outputs if it comes at different instants

  Example : double-click in a graphical IHM

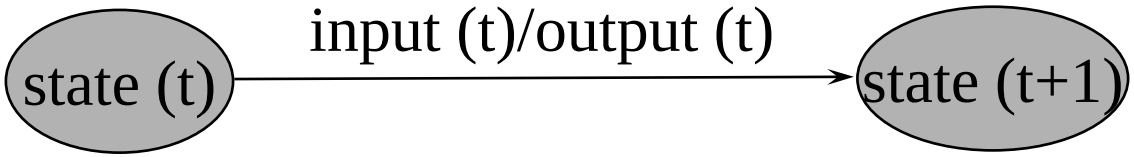
- Input at the current instant is not enough to compute the output:

  $(\neg\exists f)$ output (t) = f (input (t))

- It is necessary to take into account all the previous inputs:

  output (t) = f (input (0), input (1), ..., input (t))

Reactive programs (3/3)



- Notion of state (memory)
  - state (t) : state of the program at instant t

    summary of the program history that is useful for the future
- Inputs and current state
  - output (t+1) = f (input (t), state (t))
  - state (t+1) = g (input (t), state (t))
- Notion of transition

Principles of reactive programs

- **Concurrency :**
  - Simultaneous execution of several processes (tasks) that compete to access common resources
- **Communication :**
  - Information exchange (message sending or variable sharing) between tasks
- **Synchronisation :**
  - Waiting (rendezvous) between tasks or suspension (preemption)
- **Cooperation :**
  - Collaboration of tasks to a common objective

Real-time (RT) programs (1/2)

- Inherit the caracteristics of reactive programs
- Additionally, time matters when computing the reaction
  - output (t+1) = f (input (t), state (t), t)
  - state (t+1) = g (input (t), state (t), t)
- Example :

  timeout in communication protocols

RT programs (2/2)

- The execution follows timing constraints dictated by the environment:
  - input reading is sufficiently fast so as to not miss inputs
  - reaction is sufficiently fast to keep its meaning
- Examples:
  - avionics: ms
  - chemical reactions: minute / hour
  - mailer: day
- RT is about predicting the response times, not about getting faster programs

Remarks

- « **Soft** » **RT**: the system should not miss a deadline
  - but a small deterioration of the performance due to late response is acceptable from time to time
  - delays are respected statistically
  - examples : packet transmission in a network, cable TV
- « **Hard** » **RT**: the system must not miss a deadline
  - a late response is useless, or even wrong
  - delays may have catastrophic consequences
  - examples : plane auto-pilot, nuclear plant controller
- In a **complete system**, hard RT part is small
  - most of the system is transformational or reactive
  - Airbus 340: 5% hard RT

Asynchronous concurrency

- Basic notion: asynchronous automata
- No global clock
- Atomic actions:
  - instantaneous
  - non simultaneous
    (except rendezvous synchronisation)
- Observer point of view: interleaving of actions
- Access to shared resources:
  using mutual exclusion algorithms (e.g., Peterson algorithm)

Examples of asynchronous systems

- protocols
  - telecommunication
  - security / cryptography
- distributed systems
  - clusters & grids
  - shared virtual memory
  - distributed file systems
  - internet of things
- hardware
  - asynchronous circuits and architectures
  - multiprocessor systems
- biology: gene regulatory networks (interaction of DNA segments in a cell in function of protein concentrations)
- …

Concurrency is a difficult problem

- More rapid and even unavoidable, but much harder than sequential computing
- Many errors are possible:
  - Deadlocks
  - Race conditions
  - Loss of global consistency
- Other causes of complexity
  - Communications may fail
    (e.g., in a non reliable network)
  - Tasks/processes may fail
    (e.g., crash of a computing node)

Goals of this course

- Study base formalisms to **model** asynchronous concurrent (and possibly real-time) systems
  - Communicating automata
  - Process algebras
  - New generation formal languages
- Study some aspects of formal verification, in particular model checking
- Manipulate during lab sessions: languages and tools for modeling and verifying asynchronous concurrent systems