Modelling and verification

Designing correct concurrent and real-time systems using formal methods

Teacher

Luca Di Stefano
post-doctoral researcher, Convecs team
Inria Grenoble Rhône-Alpes
Montbonnot

luca.di-stefano@inria.fr

Course overview (1/2)

1. Introduction (18/3, 13:30-15:30)
2. Communicating automata (18/3, 15:30-17:30)

3. Process algebras (21/3 8:30-10:30)
4. Exercises on 2-3 (21/3 10:30-12:30)
5. LNT language (21/3 13:30-17:30)

6. Timed Automata (22/3 8:30-10:30)
7. Exercises on 2-3 (22/3 10:30-12:30)
8. Ex. on 5-6 + Lab: Uppaal (22/3 13:30-17:30)

Course overview (2/2)

9. Temporal logics (13/4 13:30-15:30)
10. Test generation (13/4 15:30-17:30)

11. Exercises on 9-10 (14/4 8:30-12:30)
12. Lab session on CADP (14/4 13:30-17:30)

13. Exercises + Lab on all topics (11/5 13:30-16:30)
14. Conclusions (15/4 16:30-17:30)

Final exam

- Individual homework
  - 3 practical exercises involving the tools we will see in class (worth 5 points each)
  - 1-2 “pen-and-paper” exercises (worth 5 points total)
  - Total: 20 points
- Before the deadline (TBD), you should send me
  - All files needed to “solve” the practical exercises
  - A short report containing:
    - Your solution to “pen-and-paper” exercise(s)
    - Details on the practical exercises (what do the files contain, which commands did you run, etc.)

Improving development processes using formal methods

What is a formal method

- A system development method
- Based on a formal model:
  - Rigorous system description
  - Mathematically-defined semantics
- Advantages:
  - Reference: no ambiguity
  - Some aspects of system correctness can be verified formally
  - Applications from design to implementation & test
- Applicable to both software and hardware
  - Architecture
  - Data
  - Input/outputs
  - Timed behaviour, etc.

Many formal methods

(We will see these ones)

- Petri Nets
- Process algebras
  - CCS
  - CSP
  - LNT
- Automata-based
  - Timed Automata
  - Hybrid Automata
- Quantitative FM
  - Markov chains
  - Chemical reaction networks

Why so many formal methods?

- Same situation as for programming languages
- Each formal method targets a specific domain: description of data, sequential processing, concurrency, real-time, etc.
- Each formal method has its strengths and weaknesses
- (Academia likes to explore those tradeoffs and come up with alternatives to the state of the art)

Some knowledge bases

- Wikipedia
- https://formalmethods.wikia.org/
- http://www.cs.indiana.edu/formal-methods-education/Tools
- http://i-cav.org/cavlinks/

Formal Methods:

COSTS AND BENEFITS

The initial cost of formal methods

Formal methods have the same disadvantages as any quality improvement effort:

- They require skilled engineers
- The effort put in the formal modeling does not always immediately improve the final product

But...

- They also have advantages
- Not using them also has costs and risks

1. Better quality of specifications

- 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. An easier coding phase

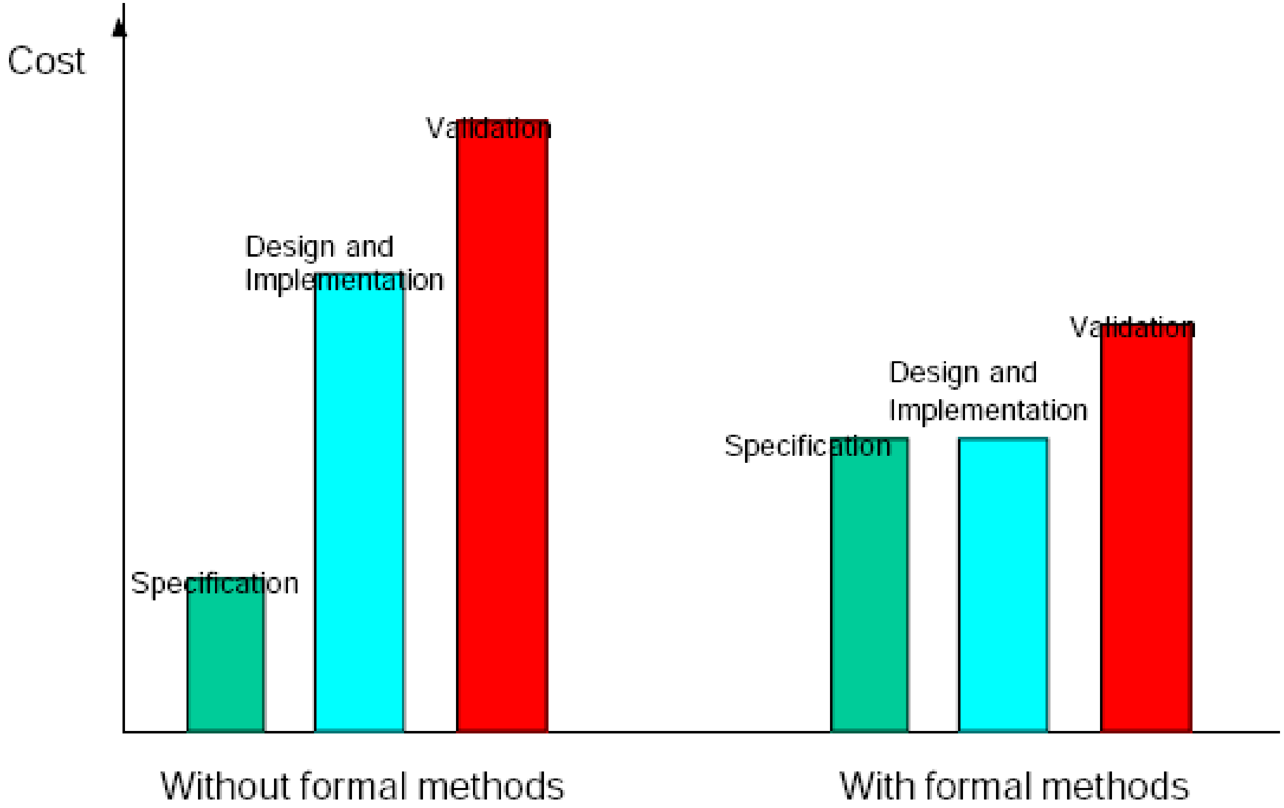
- Programmers have a clear description of what the final program must “look like”

- Ambiguities are eliminated: it is much harder for the programmer to introduce mistakes by misunderstanding the specifications

3. Earliest error detection

- Generally, the latter an error is detected, the more expensive it is to correct
- The worst errors are those detected when the product has already been delivered to the client
- With formal methods, errors are detected earlier
- A formal proof of correctness gives strong guarantees that the final product will work as intended

New distribution of effort and cost



Detecting errors earlier speeds up implementation, reduces the cost and duration of tests

Costs of NOT using Formal Methods

- Intel Pentium II `fdiv` bug (1994)
  - The “floating point division” instruction in early Pentium 2 chips gave incorrect results
  - Massive recall of the affected chips
  - 475 M$
- Ariane 5 crash (1996)
  - A conversion from a 64-bit integer to a 16-bit one caused an overflow
  - Rocket entered self-destruct
  - 370 M$

Additional outcomes

- Automated verification: automatically detect errors within the formal model (increases the number of errors discovered early)

- Code generation: synthesize source code from the specification (avoids introducing human errors in the translation)

- Test case generation: generate tests according to some criterion (decreases human effort)

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 direct the exploration) are combined

Automated verification (2/4)

What do we verify?

- Functional (or qualitative) aspects
  - Absence of deadlock (i.e., the system does not halt)
  - Determinism
  - Absence of unwanted sequences of actions
- Non-functional (or quantitative) aspects
  - Response time
  - Performance
  - Memory consumption
  - Power consumption

Automated verification (3/4)

Two main approaches to functional verification:

- Proof (or deductive verification, theorem proving): mathematically demonstrate that the property holds by application of logic rules

- Enumerative verification (brute force): enumerate and verify all possible cases

- Not mutually exclusive
- Again, several tradeoffs
  - Theorem proving may be harder to fully automate
  - Enumeration has issues with “infinite-state” systems

Automated verification (4/4)

How do we specify the properties we want to verify?

Two main approaches:

- Equivalence checking (single-language)
  - Same formalism for system and properties
- Model checking (two-language)
  - One language to describe the system
  - the other to formalise properties

Equivalence checking

- Describe both the specification S and the implementation P in the same formalism
  - Specification encodes the "good" behaviour (and is usually very compact)
  - Implementation describes how the actual system will work (and is usually larger, more detailed)
- Verify (via automated tools) that the two are equivalent: $P \sim S$
  - i.e., they "do the same things"
  - (There are multiple formalization of "equivalence")

Model Checking

- Describe the system P with a specification language
- Describe "good" behaviour with one or more logic formulas φ_1, φ_2 , … φ_n (properties)
- Show that P *models* (i.e., satisfies) all properties

$$P \vDash \varphi_i$$

Code generation and executable FMs

- A modelling language is executable if the model can be automatically transformed into executable code
- Programming languages are executable (of course!), but only some are formalized
- Some formal methods (not all of them!) are executable and are equipped with with compilers (which generate C code, for instance)
- Some modelling languages are neither formal, nor executable (e.g., parts of UML)

An informal programming language: C

6.5.16 Assignment operators

Syntax

1 *assignment-expression:*
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of
`=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=`

Constraints

2 An assignment operator shall have a modifiable lvalue as its left operand.

Semantics

3 An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue. The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand. The side effect of updating the stored value of the left operand shall occur between the previous and the next sequence point.

4 The order of evaluation of the operands is unspecified. If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined.

A formal programming language: SML

Atomic Expressions $\boxed{C \vdash atexp \Rightarrow \tau}$

$$\frac{}{C \vdash scon \Rightarrow \mathrm{type}(scon)} \tag{1}$$

$$\frac{C(longvid) = (\sigma, is) \qquad \sigma \succ \tau}{C \vdash longvid \Rightarrow \tau} \tag{2}$$

$$\frac{\langle C \vdash exprow \Rightarrow \varrho \rangle}{C \vdash \texttt{\{}\ \langle exprow \rangle\ \texttt{\}} \Rightarrow \{\}\langle + \ \varrho \rangle \text{ in Type}} \tag{3}$$

$$\frac{C \vdash dec \Rightarrow E \qquad C \oplus E \vdash exp \Rightarrow \tau \qquad \mathrm{tynames}\,\tau \subseteq T \text{ of } C}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau} \tag{4}$$

$$\frac{C \vdash exp \Rightarrow \tau}{C \vdash \texttt{(}\ exp\ \texttt{)} \Rightarrow \tau} \tag{5}$$

A formal process algebra: LNT

B.5.5 Assignment

An assignment statement terminates normally after updating the store by associating the value of its right-hand side to the assigned variable.

$$\frac{\langle V, \sigma \rangle \rightarrow_{\mathsf{e}} v}{\langle X \ \mathtt{:=}\ V, \sigma \rangle \xrightarrow{\surd}_{\mathsf{s}} \sigma \ \breve{}\ [X \leftarrow v]}$$

B.6.4 Sequential composition

The behaviour "$B_1\ ;\ B_2$" starts by executing B_1.

If B_1 terminates normally, then B_2 is executed in the store updated by B_1.

$$\frac{\langle B_1, \sigma \rangle \xrightarrow{\surd}_{\mathsf{b}} \langle B_1', \sigma' \rangle \qquad \langle B_2, \sigma' \rangle \xrightarrow{a}_{\mathsf{b}} \langle B_2', \sigma'' \rangle}{\langle B_1\ ;\ B_2, \sigma \rangle \xrightarrow{a}_{\mathsf{b}} \langle B_2', \sigma'' \rangle}$$

Rapid prototyping

- If the specification is described with an executable formal method, it can be considered as a program written in a very high-level language
- The specification can be used to quickly generate prototypes that will be shown to the client
- Possibly, all the coding can be automated
  - But beware, big specifications can have drawbacks too!

Automated test generation

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

- This approach reduces the testing effort

Examples:
- TGV (*Test Generation based on Verification*)
- GATeL (developed at CEA/LIST)
- TESTOR

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

Formal methods:

INDUSTRY IMPACT

Hardware industry

- FMs are now commonly used for circuit and architecture designs
  - Example: the PSL (*Property Specification Language*) of the Accellera consortium: http://www.accellera.org
- Essentially, every new design incorporates FMs in the signoff phase
- Some manufacturers even develop their own tools
  - Intel
  - IBM

Software industry (1/2)

FMs are not widespread in the software industry.

- They are a young subject (~50 years)
- They require theoretical skills
- They are not general: usually they are only relevent to the most complex parts of a system
- There are many of them, with different tradeoffs
  - additional effort: which parts of the systems should be treated formally? Which formal method is best suited?

Software industry (2/2)

- Distrust: initial goals were too ambitious
- Time-to-market is more important than early detection of errors
- Difficult to predict if the overhead caused by FMs will pay off in the future
- Competing techniques (e.g., software testing):
  - catch a good amount of “shallow” bugs
  - require less technical expertise

But things are changing...

FMs in the software industry: examples

- Microsoft: Verification of Windows drivers (WDF)
- Facebook: Verification of web/mobile apps (Infer)
- Amazon: Verification of AWS components (TLA+)

[Software verification] has been the Holy Grail of computer science for many decades.

But now, in some very key areas, for example driver verification, we're building tools that can do actual proofs of the software and how it works in order to guarantee the reliability.

Bill Gates, 2002

Successes in the « software » domain

- Example 1 : The SPIN model checker (Bell Labs)
  http://spinroot.com/spin/whatispin.html
  - 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

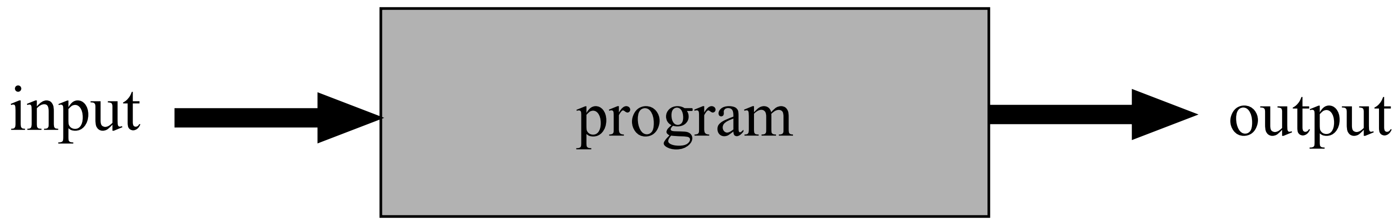
  > 200 case studies in various domains

Summary

- For specification and design, FMs are an improvement with respect of usual practices of natural language + diagrams.
- They require expertise and thus are mostly used for critical systems
  - avionics, energy plants, circuits, etc.
- Their cost (early phases) can be compensated later
  - automated coding, validation, test generation, etc.
  - This can deeply modify the traditional development cycle.
- The formal method to use must be chosen according to the nature of the problem

CONCURRENT, REACTIVE, REAL-TIME SYSTEMS

Transformational programs

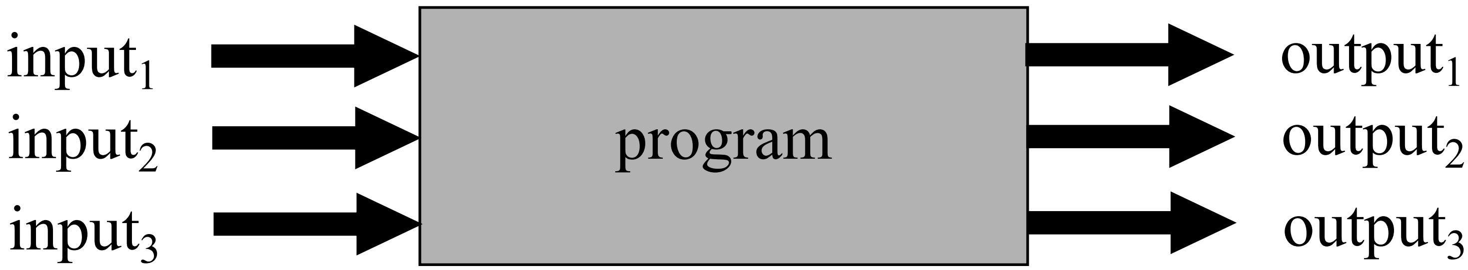


- Sequential behaviour
- Termination is normal, even expected
- Maps inputs to outputs: *output = f(input)*

Examples:

- Algorithms (sorting, classifiers, arithmetic ops, …)
- Compilers
- Command-line tools

Reactive systems (1/3)



- Cyclic behaviour
- Termination (“deadlock”) is abnormal
- Receive inputs and respond with outputs

Examples:

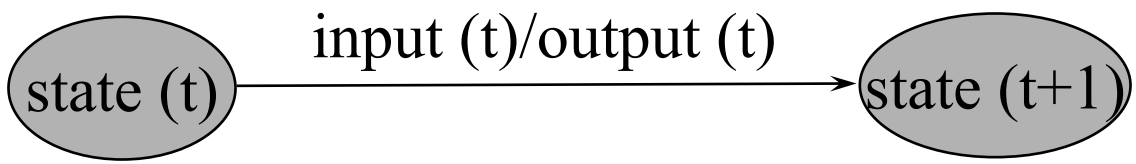
- Operating systems
- Graphical interfaces
- Servers

Reactive systems (2/3)

- The same input can produce different outputs if it comes at different instants
  - Double-click in a graphical interface
  - Request to access a shared resource
- Input is not a single value: it's a function of time
- Same for output
- The output of a reactive system must take into account all the previous inputs:

$$output(t) = f\,(input(0), \ldots, input(t))$$

Reactive systems (3/3)



- State: "summary" of the inputs
  - *state(t)* : state of the program at instant *t*
- Next output & state are affected by current input and state
  - $output(t+1) = f\,(input(t), state(t))$
  - $state(t+1) = g\,(input(t), state(t))$
- Transitions from one instant (t) to (t+1)

Principles of reactive systems

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

Asynchronous concurrency

- No global clock
- Atomic actions
  - Instantaneous
  - Non-simultaneous
- Automata may synchronize on specific actions
  - E.g. inputs and outputs
  - These actions are considered to happen simultaneously
- Observer point of view: interleaving of actions

Examples of asynchronous systems

- Protocols
  - communication
  - security / cryptography
- Distributed systems
  - clusters & grids
  - shared virtual memory
  - internet of things
- Hardware
  - asynchronous circuits and architectures
  - multiprocessor systems

Concurrency is a difficult problem

- Much harder than sequential computing
- Unavoidable
  - We want to exploit parallel computing
  - Some scenarios (e.g., networks of computers) can only be seen as concurrent systems
- Many errors are possible
  - Deadlocks
  - Race conditions, etc.
- Other causes of complexity
  - Communications may fail
  - Tasks/processes may fail, etc.

Real-time (RT) systems

- Inherit the features of reactive systems
- Furthermore, time matters
  - $output\ (t+1) = f\ (input\ (t), state\ (t), t)$
  - $state\ (t+1) = g\ (input\ (t), state\ (t), t)$
- Execution is time-constrained
  - A late reaction to some input ("missing a deadline") may be useless or even wrong

Examples

- Communication protocols with timeout
- Electronic circuits (the timing of signals is important)
- Controllers for, e.g., autonomous vehicles

Soft vs. Hard RT

- Soft RT: the system *should* not miss a deadline
  - but may do it from time to time: it can cause some degradation, but the system may recover
  - E.g., IP router
- Hard RT: the system *must* respect all deadlines
  - Failure to do so may have catastrophic consequences
  - E.g., plane autopilot
- The hard RT part of a system is usually small
  - Airbus 340: 5% hard RT

Goals of this course

- Study basic formalisms for concurrency
  - Communicating automata
  - Process algebras
  - Property languages
  - Timed extensions
- Study some aspects of formal verification (model checking, equivalence checking), and other techniques enabled by FMs (e.g. test generation)
- Lab sessions: languages and tools for modelling and verifying asynchronous concurrent systems