Domain Specific Formal Languages

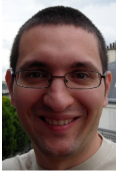
General Info & Introduction

Francesco Tiezzi

University of Camerino
`francesco.tiezzi@unicam.it`

A.A. 2017/2018

Who I am



Prof. Francesco Tiezzi

Associate Professor at University of Camerino

web: http://tiezzi.unicam.it

tel.: +39 0737 402593

e-mail: francesco.tiezzi@unicam.it

address: University of Camerino
School of Science and Technology
Computer Science Division
Polo Lodovici
Via Madonna delle Carceri, 9
62032, Camerino (MC), Italy

Schedule

MON	**TUE**	**WED**	**THU**	**FRI**
14-16		11-13		

Contents

- Domain Specific Languages (DSL)
- Brief introduction to preliminary mathematical concepts at the basis of the topic faced in the course
- From CCS to pi-calculus: syntax and semantics
- DSL for distributed systems: Dpi, Djoin, Ambient, Klaim/Klava
- DSL for service-oriented systems: COWS/SocL/CMC, CaSPiS, SOCK/Jolie, Blite/BliteC
- DSL for access control policies: FACPL
- DSL for cloud computing systems: SLAC/dSLAC, Mobica
- DSL for autonomic systems: SCEL/jRESP
- DSL for business process modelling: BPMN formalisation

Prerequisites

- Content from the FORMAL MODELLING OF SOFTWARE INTENSIVE SYSTEMS (FMSIS) course, such as
  - finite state automata
  - context-free grammars
  - inference systems
  - syntax and semantics of CCS
  - ...
- These topics will be anyway briefly illustrated at the beginning of the course

Teaching material

- Luca Aceto, Anna Ingolfsdottir, Kim Guldstrand Larsen and Jiri Srba. *Reactive Systems. Modelling, Specification and Verification*. Cambridge University Press, 2007. ISBN: 9780521875462. Additional material available at book's site: `http://rsbook.cs.aau.dk`
- Course's slides
- Lecture notes, papers and slides may be given by the teacher for studying and for exercises

Final exam

- Written test
  - on the exam date a written test takes place, it has a mixed structure: solution of exercises, and open/close answer questionnaire
  - during the course in itinere tests take place; in case they are evaluated positively, they replace the written test of the exam date
- Realisation of a project with a software tool presented during the course, or writing of a report; there is an oral discussion

The Hard Life of Programmers (and Students)

Questions?

Software-Intensive Systems

Software-Intensive Systems

Are those complex systems where software contributes essential influences to the design, construction, deployment and evolution of the system as a whole [IEEE Standard 1471]

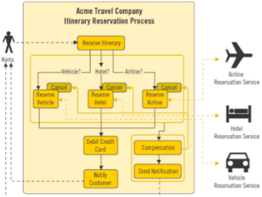
Software-Intensive Distributed Systems (SIDS)

- large-scale, decentralised, heterogeneous, highly-dynamic, open-ended, adaptive, . . .
- SIDS feature complex interactions among components
- SIDS may interact with other systems, devices, sensors, people, . . .

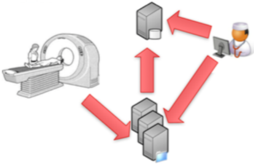
Software-Intensive Systems Everywhere

Embedded automotive systems

Robotic systems



Business processes (web services)



e-Health systems

Cloud systems

Process algebraic approach

Process Algebraic Approach to Software Intensive Systems Design

- Process algebra: theory that underpins the semantics of concurrent programming and the understanding of concurrent, distributed, and mobile systems
- It provides a natural approach to the design of those systems structuring them into a set of autonomous components that can evolve independently of each other and from time to time can *communicate* or simply *synchronize*
  - compositionality: ability to build **complex distributed systems** by combining simpler systems
  - abstraction: ability to neglect certain parts of a model
- Tools assist modeling and analysis of the various functional and non-functional aspects of those systems

SIDS as Concurrent Systems

Multiple processes (or threads) working together to achieve a common goal

- A sequential program has a single thread of control
- A concurrent program has multiple threads of control allowing it to perform multiple computations in parallel and to control multiple external activities occurring at the same time

Communication

The concurrent threads exchange information via

- indirect communication: the execution of concurrent processes proceeds on one or more processors all of which access a shared memory; care is required to deal with shared variables
- direct communication: concurrent processes are executed by running them on separate processors, threads communicate by exchanging messages

Examples of multi-threaded programs

1. windowing systems on PCs
2. embedded real-time systems, electronics, cars, telecom
3. web servers, database servers . . .
4. operating system kernel

Sequential Programming vs Concurrent Programming

Sequential Programming

- Denotational semantics: the meaning of a program is a partial function from states to states
- Nontermination is bad!
- In case of termination, the result is unique

Concurrent - Interactive - Reactive Programming

- Denotational semantics is very complicate due to nondeterminism
- Nontermination might be good!
- In case of termination, the result might not be unique

Sequential Programming vs Concurrent Programming

Sequential Programming

- Denotational semantics: the meaning of a program is a partial function from states to states
- Nontermination is bad!
- In case of termination, the result is unique

Concurrent - Interactive - Reactive Programming

- Denotational semantics is very complicate due to nondeterminism
- Nontermination might be good!
- In case of termination, the result might not be unique

Sequential Programming vs Concurrent Programming

Sequential Programming

- Denotational semantics: the meaning of a program is a partial function from states to states
- Nontermination is bad!
- In case of termination, the result is unique

Concurrent - Interactive - Reactive Programming

- Denotational semantics is very complicate due to nondeterminism
- Nontermination might be good!
- In case of termination, the result might not be unique

Sequential Programming vs Concurrent Programming

Sequential Programming

- Denotational semantics: the meaning of a program is a partial function from states to states
- Nontermination is bad!
- In case of termination, the result is unique

Concurrent - Interactive - Reactive Programming

- Denotational semantics is very complicate due to nondeterminism
- Nontermination might be good!
- In case of termination, the result might not be unique

Sequential Programming vs Concurrent Programming

Sequential Programming

- Denotational semantics: the meaning of a program is a partial function from states to states
- Nontermination is bad!
- In case of termination, the result is unique

Concurrent - Interactive - Reactive Programming

- Denotational semantics is very complicate due to nondeterminism
- Nontermination might be good!
- In case of termination, the result might not be unique

Sequential Programming vs Concurrent Programming

Sequential Programming

- Denotational semantics: the meaning of a program is a partial function from states to states
- Nontermination is bad!
- In case of termination, the result is unique

Concurrent - Interactive - Reactive Programming

- Denotational semantics is very complicate due to nondeterminism
- Nontermination might be good!
- In case of termination, the result might not be unique

SIDS as Reactive Systems

The classical denotational approach is not adequate for modelling systems such as:

- Operating systems
- Communication protocols
- Mobile phones
- Vending machines

The above systems compute by reacting to stimuli from their environment and are known as Reactive Systems; their distinguishing features are:

- Interaction (many parallel communicating processes)
- Nondeterminism (results are not necessarily unique)
- There may be no visible result (exchange of messages is used to coordinate progress)
- Nontermination is good (systems are expected to run continuously)

SIDS as Reactive Systems

The classical denotational approach is not adequate for modelling systems such as:

- Operating systems
- Communication protocols
- Mobile phones
- Vending machines

The above systems compute by reacting to stimuli from their environment and are known as Reactive Systems; their distinguishing features are:

- Interaction (many parallel communicating processes)
- Nondeterminism (results are not necessarily unique)
- There may be no visible result (exchange of messages is used to coordinate progress)
- Nontermination is good (systems are expected to run continuously)

Analysis of Reactive Systems

Even short parallel programs may be hard to analyse, thus we need to face few questions:

1. How can we develop (design) a system that “works”?
2. How do we analyse (verify) such a system?

We need appropriate theories and formal methods and tools, otherwise we will experience again:

- Intels Pentium-II bug in floating-point division unit
- Ariane-5 crash due to a conversion of 64-bit real to 16-bit integer
- . . .

Analysis of Reactive Systems

Even short parallel programs may be hard to analyse, thus we need to face few questions:

1. How can we develop (design) a system that "works"?
2. How do we analyse (verify) such a system?

We need appropriate theories and formal methods and tools, otherwise we will experience again:

- Intels Pentium-II bug in floating-point division unit
- Ariane-5 crash due to a conversion of 64-bit real to 16-bit integer
- . . .

Analysis of Reactive Systems

Even short parallel programs may be hard to analyse, thus we need to face few questions:

1. How can we develop (design) a system that "works"?
2. How do we analyse (verify) such a system?

We need appropriate theories and formal methods and tools, otherwise we will experience again:

- Intels Pentium-II bug in floating-point division unit
- Ariane-5 crash due to a conversion of 64-bit real to 16-bit integer
- . . .

Analysis of Reactive Systems

Even short parallel programs may be hard to analyse, thus we need to face few questions:

1. How can we develop (design) a system that “works”?
2. How do we analyse (verify) such a system?

We need appropriate theories and formal methods and tools, otherwise we will experience again:

- Intels Pentium-II bug in floating-point division unit
- Ariane-5 crash due to a conversion of 64-bit real to 16-bit integer
- ...

Why formal methods?

- Understanding the overall behaviour resulting from system interactions can be tricky and error-prone

Simple motivating example

Consider the code: `x = 1; y = x++ + x++;`
What is the value of `x` and `y` after its execution?

Consider the code: `g(x)=g(x-1)` with `f(x)=1;`
What is the value of `f(g(42))` after its execution?

- It is even more critical when concurrency and interactions enter the game...
- Solid mathematical foundations lay the basis for formal reasoning on systems behavior

The programmer can avoid operator ++, but

Why formal methods?

- Understanding the overall behaviour resulting from system interactions can be tricky and error-prone

Simple motivating example

Consider the code: `x = 1; y = x++ + x++;`
What is the value of `x` and `y` after its execution?

Consider the code: `g(x)=g(x-1)` with `f(x)=1;`
What is the value of `f(g(42))` after its execution?

- It is even more critical when concurrency and interactions enter the game...
- Solid mathematical foundations lay the basis for formal reasoning on systems behavior

The programmer can avoid operator ++, but

Why formal methods?

- Understanding the overall behaviour resulting from system interactions can be tricky and error-prone

Simple motivating example

Consider the code: `x = 1; y = x++ + x++;`
What is the value of `x` and `y` after its execution?

Consider the code: `g(x)=g(x-1)` with `f(x)=1;`
What is the value of `f(g(42))` after its execution?

- It is even more critical when concurrency and interactions enter the game…
- Solid mathematical foundations lay the basis for formal reasoning on systems behavior

The programmer can avoid operator ++, but

Why formal methods?

- Understanding the overall behaviour resulting from system interactions can be tricky and error-prone

Simple motivating example

Consider the code: `x = 1; y = x++ + x++;`
What is the value of `x` and `y` after its execution?

Consider the code: `g(x)=g(x-1)` with `f(x)=1;`
What is the value of `f(g(42))` after its execution?

- It is even more critical when concurrency and interactions enter the game...
- Solid mathematical foundations lay the basis for formal reasoning on systems behavior

The programmer can avoid operator ++, but

- we cannot afford to stop building complex systems
- we need to build trustworthy systems

Why formal methods?

- Understanding the overall behaviour resulting from system interactions can be tricky and error-prone

Simple motivating example

Consider the code: `x = 1; y = x++ + x++;`
What is the value of `x` and `y` after its execution?

Consider the code: `g(x)=g(x-1)` with `f(x)=1;`
What is the value of `f(g(42))` after its execution?

- It is even more critical when concurrency and interactions enter the game...
- Solid mathematical foundations lay the basis for formal reasoning on systems behavior

The programmer can avoid operator ++, but

- we cannot afford to stop building complex systems
- we need to build trustworthy systems

Why formal methods?

- Understanding the overall behaviour resulting from system interactions can be tricky and error-prone

Simple motivating example

Consider the code: `x = 1; y = x++ + x++;`
What is the value of `x` and `y` after its execution?

Consider the code: `g(x)=g(x-1)` with `f(x)=1;`
What is the value of `f(g(42))` after its execution?

- It is even more critical when concurrency and interactions enter the game...
- Solid mathematical foundations lay the basis for formal reasoning on systems behavior

The programmer can avoid operator ++, but

- we cannot afford to stop building complex systems
- we need to build trustworthy systems

Why formal methods?

- Understanding the overall behaviour resulting from system interactions can be tricky and error-prone

Simple motivating example

Consider the code: `x = 1; y = x++ + x++;`
What is the value of `x` and `y` after its execution?

Consider the code: `g(x)=g(x-1)` with `f(x)=1;`
What is the value of `f(g(42))` after its execution?

- It is even more critical when concurrency and interactions enter the game...
- Solid mathematical foundations lay the basis for formal reasoning on systems behavior

The programmer can avoid operator ++, but

- we cannot afford to stop building complex systems
- we need to build trustworthy systems

Formal Methods for Reactive Systems

To deal with reactive systems and guarantee their correct behaviour in all possible environments, we need:

1. To study mathematical models for the formal description and analysis of concurrent programs

2. To devise formal languages for the specification of the possible behaviour of parallel and reactive systems

   Each language comes equipped with **syntax** & **semantics**

   - **Syntax**: defines legal programs (grammar based)
   - **Semantics**: defines meaning, behavior, errors (formally)

3. To develop verification tools and implementation techniques underlying them

Domain Specific Formal Languages

Why do we need a new language and techniques for each specific application domain?

Systems must be specified as naturally as possible

- distinctive aspects of the domain are first-class citizens
  ⇒ intuitive/concise spec., no encodings
- high-level abstract models ⇒ feasible analysis
- analysis results are in terms of system features, not their low-level representation ⇒ feedbacks

Process Algebras Approach

- The chosen abstraction for reactive systems is the notion of processes
- Systems evolution is based on **process transformation**: a process performs an action and becomes another process
- Everything is (or can be viewed as) a process: buffers, shared memory, tuple spaces, senders, receivers, . . . are all processes
- Labelled Transition Systems (LTSs) describe processes behaviour, and permit modelling directly systems interaction

Process Algebras Approach

- The chosen abstraction for reactive systems is the notion of processes
- Systems evolution is based on **process transformation**: a process performs an action and becomes another process
- Everything is (or can be viewed as) a process: buffers, shared memory, tuple spaces, senders, receivers, . . . are all processes
- Labelled Transition Systems (LTSs) describe processes behaviour, and permit modelling directly systems interaction

Before Domain Specific Formal Languages. . .

. . . a recap of CCS