Fundamentals of Reactive Systems

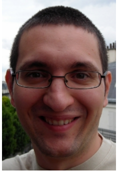
General Info & Introduction

Francesco Tiezzi

University of Camerino
francesco.tiezzi@unicam.it

A.A. 2018/2019

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
9-11		9-11		

Contents

- Introductory concepts on reactive systems
- Preliminary mathematical concepts
- Semantics of regular expressions
- Semantics of the operators of the process algebras CCS, CSP and ACP
- Main behavioural equivalences, weak and strong
- (A hint to) Hennessy-Milner Logic and ACTL
- Software tools for the automatic verification of software-intensive systems modelled by means of process algebras: TAPAs . . . and Maude, muCRL2, etc.

Teaching material

- Rocco De Nicola. *A gentle introduction to Process Algebras.* Notes obtained by the restructuring of two entries (Process Algebras - Behavioural Equivalences) of Encyclopedia of Parallel Computing, David A. Padua (Ed.). Springer 2011; pp. 120-127 and pp. 1624-1636
- 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
- TAPAs documentation `http://rap.dsi.unifi.it/tapas/`
- Maude documentation `http://maude.cs.illinois.edu`
- mCRL2 documentation `http://www.mcrl2.org/`

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 using a specification/verification software tool, or writing of a report, with a presentation

The Hard Life of Programmers (and Students)

Questions?

Reactive 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

Why Concurrent/Distributed Systems

1. Performance: To gain from multiprocessing hardware (parallelism)
2. Distribution: Some problems require a distributed solution, e.g. client-server systems on one machine and the database on a central server machine
3. Ease of programming: Some problems are more naturally solved by concurrent programs
4. Increased application throughput: an I/O call need only to block one thread
5. Increased application responsiveness: High priority threads for user requests
6. More appropriate structure: For programs which interact with the environment, control multiple activities and handle multiple events

Do I need to check reactive systems?

Programming them is error prone

- Soviet nuclear false alarm incident (1983)
  [fault in sw for missile detections]
- Therac-25 radiation overdose (1985-1987)
  [sw interlock fault due to a race condition]
- MIM-104 Patriot Missile Clock Drift (1991)
  [a sw fault in the system's clock]
- Explosion of the Ariane 5 (1996)
  [self-destruction was triggered by an overflow error]
- North America blackout (2003)
  [race condition caused an alarm system failure]
- Mars Rover problems (2004)
  [interaction among concurrent tasks caused periodic sw resets]
- . . . for sure you have experienced deadlock on your machine and pressed restart (even if you have a Mac)

Do I need to check reactive systems?

Programming them is error prone

- Soviet nuclear false alarm incident (1983)
  [fault in sw for missile detections]
- Therac-25 radiation overdose (1985-1987)
  [sw interlock fault due to a race condition]
- MIM-104 Patriot Missile Clock Drift (1991)
  [a sw fault in the system's clock]
- Explosion of the Ariane 5 (1996)
  [self-destruction was triggered by an overflow error]
- North America blackout (2003)
  [race condition caused an alarm system failure]
- Mars Rover problems (2004)
  [interaction among concurrent tasks caused periodic sw resets]
- ... for sure you have experienced deadlock on your machine and pressed restart (even if you have a Mac)

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

Analysis

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

Analysis

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

Analysis

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

Analysis

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

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

Process Algebraic Approach

- The chosen abstraction for reactive systems is the notion of processes: everything is (or can be viewed as) a process: buffers, shared memory, senders, receivers, ... are all processes
- Systems evolution is based on **process transformation**: a process performs an action and becomes another process
- A natural approach to the design of those systems structuring them into a set of components that can evolve independently and *communicate/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
- Labelled Transition Systems (LTSs) describe processes behaviour, and permit modelling directly systems interaction

Process Algebraic Approach

- The chosen abstraction for reactive systems is the notion of processes: everything is (or can be viewed as) a process: buffers, shared memory, senders, receivers, ... are all processes
- Systems evolution is based on **process transformation**: a process performs an action and becomes another process
- A natural approach to the design of those systems structuring them into a set of components that can evolve independently and *communicate/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
- Labelled Transition Systems (LTSs) describe processes behaviour, and permit modelling directly systems interaction

Internal and External Actions

Labelled Transition Systems

Transition Labelled Graph: a transition between states is labelled by the action inducing the transition from one state to another

Actions

An elementary action represents the *atomic* (non-interruptible) abstract step of a computation that is performed by a system

Actions represent various activities of concurrent systems:

- Sending a message
- Receiving a message
- Updating values
- Synchronizing with other processes ...

We have two main types of atomic actions:

- Visible Actions
- Internal Actions (τ)

Internal and External Actions

Labelled Transition Systems

Transition Labelled Graph: a transition between states is labelled by the action inducing the transition from one state to another

Actions

An elementary action represents the *atomic* (non-interruptible) abstract step of a computation that is performed by a system

Actions represent various activities of concurrent systems:

- Sending a message
- Receiving a message
- Updating values
- Synchronizing with other processes . . .

We have two main types of atomic actions:

- Visible Actions
- Internal Actions (τ)

Internal and External Actions

Labelled Transition Systems

Transition Labelled Graph: a transition between states is labelled by the action inducing the transition from one state to another

Actions

An elementary action represents the *atomic* (non-interruptible) abstract step of a computation that is performed by a system

Actions represent various activities of concurrent systems:

- Sending a message
- Receiving a message
- Updating values
- Synchronizing with other processes ...

We have two main types of atomic actions:

- Visible Actions
- Internal Actions (τ)

Why operators for describing systems

How can we describe very large automata or LTSs?

As a table?

Rows and columns are labelled by states, entries are either empty or marked with a set of actions

As a listing of triples?

$\rightarrow \; = \; \{(q_0, a, q_1), (q_0, a, q_2), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, q_3), (q_2, \tau, q_4)\}$

As a more compact listing of triples?

$\rightarrow \; = \; \{(q_0, a, \{q_1, q_2\}), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, \{q_3, q_4\})\}$

As XML?

```
<lts><ar><st>q0</st><lab>a</lab><st>q1</st></ar>...</lts>
```

Why operators for describing systems

How can we describe very large automata or LTSs?

As a table?

Rows and columns are labelled by states, entries are either empty or marked with a set of actions

As a listing of triples?

$\rightarrow \; = \; \{(q_0, a, q_1), (q_0, a, q_2), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, q_3), (q_2, \tau, q_4)\}$

As a more compact listing of triples?

$\rightarrow \; = \; \{(q_0, a, \{q_1, q_2\}), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, \{q_3, q_4\})\}$

As XML?

```
<lts><ar><st>q0</st><lab>a</lab><st>q1</st></ar>...</lts>
```

Why operators for describing systems

How can we describe very large automata or LTSs?

As a table?

Rows and columns are labelled by states, entries are either empty or marked with a set of actions

As a listing of triples?

$\rightarrow \; = \; \{(q_0, a, q_1), (q_0, a, q_2), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, q_3), (q_2, \tau, q_4)\}$

As a more compact listing of triples?

$\rightarrow \; = \; \{(q_0, a, \{q_1, q_2\}), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, \{q_3, q_4\})\}$

As XML?

`<lts><ar><st>q0</st><lab>a</lab><st>q1</st></ar>...</lts>`

Why operators for describing systems

How can we describe very large automata or LTSs?

As a table?

Rows and columns are labelled by states, entries are either empty or marked with a set of actions

As a listing of triples?

$\rightarrow \; = \; \{(q_0, a, q_1), (q_0, a, q_2), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, q_3), (q_2, \tau, q_4)\}$

As a more compact listing of triples?

$\rightarrow \; = \; \{(q_0, a, \{q_1, q_2\}), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, \{q_3, q_4\})\}$

As XML?

`<lts><ar><st>q0</st><lab>a</lab><st>q1</st></ar>...</lts>`

Why operators for describing systems

How can we describe very large automata or LTSs?

As a table?

Rows and columns are labelled by states, entries are either empty or marked with a set of actions

As a listing of triples?

$$\rightarrow \ = \ \{(q_0, a, q_1), (q_0, a, q_2), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, q_3), (q_2, \tau, q_4)\}$$

As a more compact listing of triples?

$$\rightarrow \ = \ \{(q_0, a, \{q_1, q_2\}), (q_1, b, q_3), (q_1, c, q_4), (q_2, \tau, \{q_3, q_4\})\}$$

As XML?

```
<lts><ar><st>q0</st><lab>a</lab><st>q1</st></ar>...</lts>
```

Why operators for describing systems

Linguistic aspects are important!

The previous solutions are ok for machines . . . not for humans

Are prefix and sum operators sufficient?

They are ok to describe small finite systems

- p = a.b.(c+d)
- q = a.(b.c+b.d)
- r = a.b.c+a.c.d

But additional operators are needed

- to design systems in a structured way (e.g. $p \mid q$)
- to model systems interaction
- to abstract from details
- to represent infinite systems

Why operators for describing systems

Linguistic aspects are important!

The previous solutions are ok for machines . . . not for humans

Are prefix and sum operators sufficient?

They are ok to describe small finite systems

- p = a.b.(c+d)
- q = a.(b.c+b.d)
- r = a.b.c+a.c.d

But additional operators are needed

- to design systems in a structured way (e.g. $p \mid q$)
- to model systems interaction
- to abstract from details
- to represent infinite systems

Why operators for describing systems

Linguistic aspects are important!

The previous solutions are ok for machines . . . not for humans

Are prefix and sum operators sufficient?

They are ok to describe small finite systems

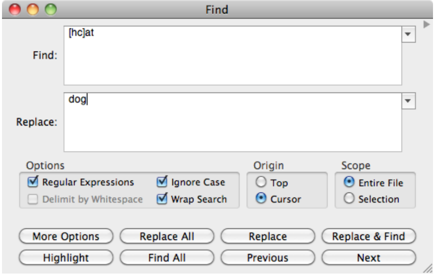
- p = a.b.(c+d)
- q = a.(b.c+b.d)
- r = a.b.c+a.c.d

But additional operators are needed

- to design systems in a structured way (e.g. $p \mid q$)
- to model systems interaction
- to abstract from details
- to represent infinite systems

A motivating example: regular expressions

Commonly used for searching and manipulating text based on patterns



Example

Regular expression: `[hc]at` $\Rightarrow$ $(h + c); a; t$
Text: `the cat eats the bat's hat rather than the rat`
Matches: `cat, hat`

A motivating example: regular expressions

Regular expressions

Commonly used for:

- searching and manipulating text based on patterns
- representing regular languages in a compact form
- describing sequences of actions that a system can execute

- Regular expressions as a simple programming language
  - Programming constructs: sequence, choice, iteration, stop
- We define the semantics of regular expressions by means of the Structural Operational Semantics approach

A motivating example: regular expressions

Regular expressions

Commonly used for:

- searching and manipulating text based on patterns
- representing regular languages in a compact form
- describing sequences of actions that a system can execute

- Regular expressions as a simple programming language
  - Programming constructs: sequence, choice, iteration, stop
- We define the semantics of regular expressions by means of the Structural Operational Semantics approach

Before syntax and semantics. . .

. . . a few preliminaries