Formal Techniques for Software Engineering: Models for Concurrency Semantics

Rocco De Nicola

IMT Institute for Advanced Studies, Lucca
`rocco.denicola@imtlucca.it`

June 2013

Lesson 6

What is Concurrent Programming

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 perform multiple computations in parallel and to control multiple external activities which occur 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 ensure exclusive access to shared variables
- direct communication: concurrent processes are executed by running them on separate processors, threads communicate by exchanging messages.

Why Concurrent Programming

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 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

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

Do I need to know about concurrent programming?

Concurrency is error prone

- Therac - 25 computerised radiation therapy machine: Concurrent programming errors contributed to accidents causing deaths and serious injuries.
- Mars Rover: Problems with interaction between concurrent tasks caused periodic software resets reducing availability for exploration.
- ....
- For sure you have experienced deadlock on your machine and pressed restart (even if you have a Mac)

Developing a Concurrent Solution to a Simple Problem

A Simple Problem

Let f a (computationally expensive) function from integers to integers.

- A positive zero for f is a positive integer n such that $f(n) = 0$
- A negative zero for f is a negative integer z such that $f(z) = 0$

Our Goal

We want to write a program that terminates if and only if the total function f has a positive or negative zero and proceeds indefinitely otherwise.

A Brilliant Idea

To speed up we decide to run in parallel two programs: one looking for a positive zero and the other for a negative zero

A Simple Problem

Let f a (computationally expensive) function from integers to integers.

- A positive zero for f is a positive integer n such that $f(n) = 0$
- A negative zero for f is a negative integer z such that $f(z) = 0$

Our Goal

We want to write a program that terminates if and only if the total function f has a positive or negative zero and proceeds indefinitely otherwise.

A Brilliant Idea

To speed up we decide to run in parallel two programs: one looking for a positive zero and the other for a negative zero

A Simple Problem

Let f a (computationally expensive) function from integers to integers.

- A positive zero for f is a positive integer n such that $f(n) = 0$
- A negative zero for f is a negative integer z such that $f(z) = 0$

Our Goal

We want to write a program that terminates if and only if the total function f has a positive or negative zero and proceeds indefinitely otherwise.

A Brilliant Idea

To speed up we decide to run in parallel two programs: one looking for a positive zero and the other for a negative zero

Attempt 1

We write S1 that looks for a positive zero:

```
S1=    found=false; n=0;
       while(!found) { n++; found=(f(n)==0); }
```

Attempt 1

We write S1 that looks for a positive zero:

```
S1=   found=false; n=0;
      while(!found) { n++; found=(f(n)==0); }
```

By cut-and-paste from S1 we write S2 that looks for a negative zero:

```
S2=   found=false; z=0;
      while(!found) { z--; found=(f(z)==0); }
```

Attempt 1

We write S1 that looks for a positive zero:

```
S1=   found=false; n=0;
      while(!found) { n++; found=(f(n)==0); }
```

By cut-and-paste from S1 we write S2 that looks for a negative zero:

```
S2=   found=false; z=0;
      while(!found) { z--; found=(f(z)==0); }
```

And we run `S1` and `S2` in parallel:

```
S1 || S2
```

Attempt 1

We write S1 that looks for a positive zero:

```
S1=   found=false; n=0;
      while(!found) { n++; found=(f(n)==0); }
```

By cut-and-paste from S1 we write S2 that looks for a negative zero:

```
S2=   found=false; z=0;
      while(!found) { z--; found=(f(z)==0); }
```

And we run `S1` and `S2` in parallel:

```
S1 || S2
```

Let f have a positive zero and not a negative one.
If `S1` terminates before S2 starts, the latter sets found to false and looks indefinitely for the nonexisting zero.

Rethinking 1

The problem is due to the fact that found is initialised to false twice.

LESSON 1

USING SHARED VARIABLES MAY LEAD TO PROBLEM

Attempt 2 (found is initialised only once)

The problem is due to the fact that found is initialised to false twice.

```
                    found=false; (R1 || R2)
where
R1=   n=0; while(!found) { n++; found=(f(n)==0); }

R2=   z=0; while(!found) { z--; found=(f(z)==0); }
```

If f has (again) only a positive zero assume that:

1. R2 is preempted when entering the while body (before z--)
2. R1 runs and finds a (positive) zero
3. R2 gets the CPU back

When R2 restarts it executes the while body and may set found to false. The program then would not terminate because it would look for a non existing negative zero.

Attempt 2 (found is initialised only once)

The problem is due to the fact that found is initialised to false twice.

```
                    found=false; (R1 || R2)
where
R1=   n=0; while(!found) { n++; found=(f(n)==0); }

R2=   z=0; while(!found) { z--; found=(f(z)==0); }
```

If f has (again) only a positive zero assume that:

1. R2 is preempted when entering the while body (before `z--`)
2. R1 runs and finds a (positive) zero
3. R2 gets the CPU back

When R2 restarts it executes the while body and may set found to false. The program then would not terminate because it would look for a non existing negative zero.

Rethinking 2

The problem is due to the fact that found is set to false after it has already been assigned true.

LESSON 2

NO ASSUMPTION ABOUT THE MOMENT A PROGRAM IS INTERRUPTED CAN BE MADE (It can only be programmed).

Attempt 3 ("unnecessary" assignments are removed)

The problem is due to the fact that found is set to false after it has already been assigned true.

```
                    found=false; (T1 || T2)
where

T1=   n=0; while(!found) { n++; if (f(n)==0) found=true; }

T2=   z=0; while(!found) { z--; if (f(z)==0) found=true; }
```

Let f have only a positive zero.
Assume that T2 gets the CPU to keep it until it terminates. Since this will never happen, T1 will never get the chance to find its zero.

Attempt 3 ("unnecessary" assignments are removed)

The problem is due to the fact that found is set to false after it has already been assigned true.

```
                    found=false; (T1 || T2)
where

T1=   n=0; while(!found) { n++; if (f(n)==0) found=true; }

T2=   z=0; while(!found) { z--; if (f(z)==0) found=true; }
```

Let f have only a positive zero.
Assume that T2 gets the CPU to keep it until it terminates. Since this will never happen, T1 will never get the chance to find its zero.

Rethinking 3

The problem is due to non-fair scheduling policies.

LESSON 3

NO ASSUMPTION CAN BE MADE ON THE SCHEDULING POLICY OF THE CPU.

Attempt 4 (token passing fairness)

The problem is due to non-fair scheduling policies.

```
               turn=1; found=false; (Q1 || Q2)
where
Q1=   n=0; while(!found) {
                wait turn==1 then {
                    turn=2; n++; if (f(n)==0) found=true; } }

Q2=   z=0; while(!found) {
                wait turn==2 then {
                    turn=1; z--; if (f(z)==0) found=true; } }
```

If Q1 finds a zero and stops when Q2 has already set turn to 1, Q2 would be blocked by the wait command because the value of turn cannot be changed.

Attempt 4 (token passing fairness)

The problem is due to non-fair scheduling policies.

```
                turn=1; found=false; (Q1 || Q2)
where
Q1=   n=0; while(!found) {
                wait turn==1 then {
                    turn=2; n++; if (f(n)==0) found=true; } }

Q2=   z=0; while(!found) {
                wait turn==2 then {
                    turn=1; z--; if (f(z)==0) found=true; } }
```

If Q1 finds a zero and stops when Q2 has already set turn to 1, Q2 would be blocked by the wait command because the value of turn cannot be changed.

Rethinking 4

The program may not terminate, waiting for an impossible event.

LESSON 4

ON TERMINATION CARE IS NEEDED FOR ALL PROCESSES.

Attempt 5 (pass the token before terminating)

The program may not terminate, waiting for an impossible event.

Is it a correct solution?

```
    turn=1; found=false; ( {P1; turn=2;} || {P2; turn=1;} )
where

P1=   n=0; while(!found) {
                wait turn==1 then {
                    turn=2; n++;
                    if (f(n)==0) found=true; } }

P2=   z=0; while(!found) {
                wait turn==2 then {
                    turn=1; z--;
                    if (f(z)==0) found=true; } }
```

Sequential Programming vs Concurrent Programming

Classical Sequential Programming

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

Concurrent - Interactive - Reactive Programming

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

Sequential Programming vs Concurrent Programming

Classical Sequential Programming

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

Concurrent - Interactive - Reactive Programming

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

Sequential Programming vs Concurrent Programming

Classical Sequential Programming

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

Concurrent - Interactive - Reactive Programming

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

Sequential Programming vs Concurrent Programming

Classical Sequential Programming

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

Concurrent - Interactive - Reactive Programming

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

Sequential Programming vs Concurrent Programming

Classical Sequential Programming

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

Concurrent - Interactive - Reactive Programming

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

Sequential Programming vs Concurrent Programming

Classical Sequential Programming

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

Concurrent - Interactive - Reactive Programming

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

Programming Reactive System

The classical denotational approach is not sufficient 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)

Programming Reactive System

The classical denotational approach is not sufficient 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)

Programming Reactive System

The classical denotational approach is not sufficient 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)

Programming Reactive System

The classical denotational approach is not sufficient 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)

Programming Reactive System

The classical denotational approach is not sufficient 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)

Programming Reactive System

The classical denotational approach is not sufficient 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)

Programming Reactive System

The classical denotational approach is not sufficient 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)

Programming Reactive System

The classical denotational approach is not sufficient 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)

Programming Reactive System

The classical denotational approach is not sufficient 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)

Programming Reactive System

The classical denotational approach is not sufficient 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 analyze, thus we need to face few questions:

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

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

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

Analysis of Reactive Systems

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

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

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

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

Analysis of Reactive Systems

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

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

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

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

Analysis of Reactive Systems

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

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

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

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

Analysis of Reactive Systems

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

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

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

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

Analysis of Reactive Systems

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

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

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

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

Formal Methods for Reactive Systems

To deal with reactive systems and guarantee their correct behavior in all possible environment, 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.
3. To develop verification tools and implementation techniques underlying them.

Formal Methods for Reactive Systems

To deal with reactive systems and guarantee their correct behavior in all possible environment, 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.
3. To develop verification tools and implementation techniques underlying them.

Formal Methods for Reactive Systems

To deal with reactive systems and guarantee their correct behavior in all possible environment, 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.
3. To develop verification tools and implementation techniques underlying them.

This part of the course

We shall see different theories of special kind of reactive systems and their applications.

The theories aim at supporting: Design, Specification and Verification (possibly automatic and compositional) of reactive systems.

Important Questions:

- What is the most abstract view of a reactive system (process)?
- Does it capture their relevant properties?
- Is it compositional?

This part of the course

We shall see different theories of special kind of reactive systems and their applications.

The theories aim at supporting: Design, Specification and Verification (possibly automatic and compositional) of reactive systems.

Important Questions:

- What is the most abstract view of a reactive system (process)?
- Does it capture their relevant properties?
- Is it compositional?

This part of the course

We shall see different theories of special kind of reactive systems and their applications.

The theories aim at supporting: Design, Specification and Verification (possibly automatic and compositional) of reactive systems.

Important Questions:

- What is the most abstract view of a reactive system (process)?
- Does it capture their relevant properties?
- Is it compositional?

Our 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 (LTS) describe process behaviour, and permit modelling directly systems interaction.

Outline of the lectures

Outline of the lectures

1. Labelled Transition Systems as Concurrency Models
2. Operators for Interaction, Nondeterminism and Concurrency
3. Process Calculi and their semantics
4. A Calculus of Communicating Systems
5. Modal and Temporal Logics
6. Tools for Systems Specification and Verification

Outline of the lectures

1. Labelled Transition Systems as Concurrency Models
2. Operators for Interaction, Nondeterminism and Concurrency
3. Process Calculi and their semantics
4. A Calculus of Communicating Systems
5. Modal and Temporal Logics
6. Tools for Systems Specification and Verification

Outline of the lectures

1. Labelled Transition Systems as Concurrency Models
2. Operators for Interaction, Nondeterminism and Concurrency
3. Process Calculi and their semantics
4. A Calculus of Communicating Systems
5. Modal and Temporal Logics
6. Tools for Systems Specification and Verification

Outline of the lectures

1. Labelled Transition Systems as Concurrency Models
2. Operators for Interaction, Nondeterminism and Concurrency
3. Process Calculi and their semantics
4. A Calculus of Communicating Systems
5. Modal and Temporal Logics
6. Tools for Systems Specification and Verification

Outline of the lectures

1. Labelled Transition Systems as Concurrency Models
2. Operators for Interaction, Nondeterminism and Concurrency
3. Process Calculi and their semantics
4. A Calculus of Communicating Systems
5. Modal and Temporal Logics
6. Tools for Systems Specification and Verification

Bibliography

- Apt K.R., Olderog E.-R., *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1997.
  I took from here the first example of these lectures.
- Fokkink Wan, *Introduction to Process Algebra*, Springer, 2000.
  A gentle introduction to ACP.
- Milner R., *Communication and Concurrency*, Prentice Hall, 1989.
  The classical book on CCS and Bisimulation.
- Roscoe A.W., *The Theory and Practice of Concurrency*, Prentice Hall, 1998.
  A good book on TCSP and the failure Model.
- Bowman H. and Gomez R., *Concurrency Theory: Calculi and Automata for Modelling Untimed and Timed Concurrent Systems*, Springer, 2006.
  A new book on concurrency theory based on LOTOS.

Bibliography ctd.

- Baeten J.C.M. and Weijland W.P., *Process Algebra*, Cambridge University Press, 1990.
  The first book on ACP and Branching Bisimulation.

- Hennessy M., *Algebraic theory of processes*, Springer-Verlag, 2001.
  A simple introduction to Algebraic, Denotational and Operational Semantics of processes based on Testing Equivalence.

- Van Glabbeek R.J., *The Linear Time - Branching Time Spectrum I*. The Semantics of Concrete, Sequential Processes*, Handbook on Process Algebras, North Holland, 2001.
  A good overview of behavioral equivalences over LTS.

- Sangiorgi D. and Walker D., *PI-Calculus: A Theory of Mobile Processes*, Cambridge University Press, 2001.
  THE book on π-calculus.

- For Klaim see: `http://music.dsi.unifi.it`

Operational Semantics for Concurrent Processes

Systems behaviour is described by associating to each program a behaviour represented as a transition graph.

Two main models (or variants thereof) have been used:

Kripke Structures

State Labelled Graphs: States are labelled with the properties that are considered relevant (e.g. the value of - the relation between - some variables)

Labelled Transition Systems

Transition Labelled Graph: Transition between states are labelled with the action that induces the transition from one state to another.

In this lectures, we shall mainly rely on **Labelled Transition Systems** and actions will play an important role

Operational Semantics for Concurrent Processes

Systems behaviour is described by associating to each program a behaviour represented as a transition graph.

Two main models (or variants thereof) have been used:

Kripke Structures

State Labelled Graphs: States are labelled with the properties that are considered relevant (e.g. the value of - the relation between - some variables)

Labelled Transition Systems

Transition Labelled Graph: Transition between states are labelled with the action that induces the transition from one state to another.

In this lectures, we shall mainly rely on **Labelled Transition Systems** and actions will play an important role

Operational Semantics for Concurrent Processes

Systems behaviour is described by associating to each program a behaviour represented as a transition graph.

Two main models (or variants thereof) have been used:

Kripke Structures

State Labelled Graphs: States are labelled with the properties that are considered relevant (e.g. the value of - the relation between - some variables)

Labelled Transition Systems

Transition Labelled Graph: Transition between states are labelled with the action that induces the transition from one state to another.

In this lectures, we shall mainly rely on **Labelled Transition Systems** and actions will play an important role

Operational Semantics for Concurrent Processes

Systems behaviour is described by associating to each program a behaviour represented as a transition graph.

Two main models (or variants thereof) have been used:

Kripke Structures

State Labelled Graphs: States are labelled with the properties that are considered relevant (e.g. the value of - the relation between - some variables)

Labelled Transition Systems

Transition Labelled Graph: Transition between states are labelled with the action that induces the transition from one state to another.

In this lectures, we shall mainly rely on **Labelled Transition Systems** and actions will play an important role

Finite State Automata

Definition

A *finite state automaton* M is a 5-tuple
$M = (Q, A, \rightarrow, q_0, F)$ *where*

- Q is a finite set of states
- A is the alphabet
- $\rightarrow \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$ is the transition relation
- $q_0 \in Q$ is a special state called initial state,
- $F \subseteq Q$ is the set of (final states)

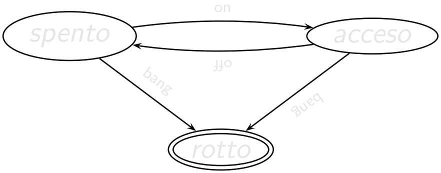


Figure: Finite state automaton

Finite State Automata

Definition

A *finite state automaton* M is a 5-tuple
$M = (Q, A, \rightarrow, q_0, F)$ *where*

- Q is a finite set of states
- A is the alphabet
- $\rightarrow \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$ is the transition relation
- $q_0 \in Q$ is a special state called initial state,
- $F \subseteq Q$ is the set of (final states)

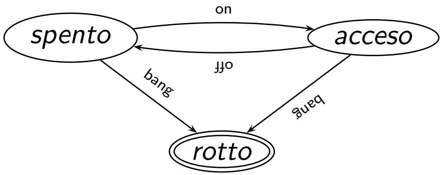


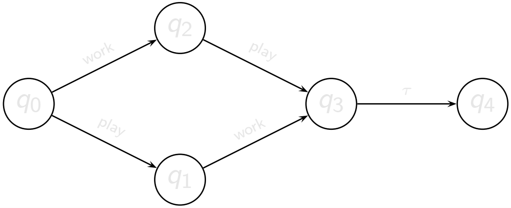
Figure: Finite state automaton

Labelled Transition Systems

Definition

A Labelled Transition System S is a 4-tuple $S = (Q, A, \rightarrow, q_0)$ *where* :

- Q is a set of states
- A is a finite set of actions
- $\rightarrow \subseteq Q \times A \times Q$ is a ternary relation called transition relation it is often written $q \xrightarrow{a} q'$ instead of $(q, a, q') \in \rightarrow$
- $q_0 \in Q$ is a special state called initial state.



If initial states are not relevant (or known) LTSs are triples $(Q, A, \rightarrow)$ ...

Labelled Transition Systems

Definition

A Labelled Transition System S is a 4-tuple $S = (Q, A, \rightarrow, q_0)$ *where* :

- Q is a set of states
- A is a finite set of actions
- $\rightarrow \subseteq Q \times A \times Q$ is a ternary relation called transition relation it is often written $q \xrightarrow{a} q'$ instead of $(q, a, q') \in \rightarrow$
- $q_0 \in Q$ is a special state called initial state.

$$\begin{tikzcd}
 & q_2 \arrow[dr, "\text{play}"] & & \\
q_0 \arrow[ur, "\text{work}"] \arrow[dr, "\text{play}"'] & & q_3 \arrow[r, "\tau"] & q_4 \\
 & q_1 \arrow[ur, "\text{work}"'] & &
\end{tikzcd}$$

If initial states are not relevant (or known) LTSs are triples $(Q, A, \rightarrow)$ ...

Labelled Transition Systems

Definition

A Labelled Transition System S is a 4-tuple $S = (Q, A, \rightarrow, q_0)$ *where* :

- Q is a set of states
- A is a finite set of actions
- $\rightarrow \subseteq Q \times A \times Q$ is a ternary relation called transition relation it is often written $q \xrightarrow{a} q'$ instead of $(q, a, q') \in\rightarrow$
- $q_0 \in Q$ is a special state called initial state.

$$\begin{tikzcd}
 & q_2 \arrow[dr, "play"] & & \\
q_0 \arrow[ur, "work"] \arrow[dr, "play"'] & & q_3 \arrow[r, "\tau"] & q_4 \\
 & q_1 \arrow[ur, "work"'] & &
\end{tikzcd}$$

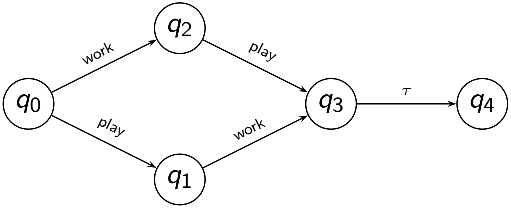
If initial states are not relevant (or known) LTSs are triples $(Q, A, \rightarrow)$ . . .

A Simple Example

Example (Bill-Ben)

$S = (Q, A, \rightarrow)$ where:

- $Q = \{\ q_0,\ q_1,\ q_2,\ q_3,\ q_4\ \}$
- $A = \{\ play,\ work,\ \tau\ \}$
- $\rightarrow = \{(q_0, play, q_1),\ (q_0, work, q_2),\ (q_1, work, q_3),\ (q_2, play, q_3),\ (q_3, \tau, q_4)\}$



$$\begin{tikzcd}
 & q_2 \arrow[dr, "play"] & & \\
q_0 \arrow[ur, "work"] \arrow[dr, "play"'] & & q_3 \arrow[r, "\tau"] & q_4 \\
 & q_1 \arrow[ur, "work"'] & &
\end{tikzcd}$$

Internal and External Actions

An elementary action of a system represents the atomic (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the other.

Actions represent various activities of concurrent systems:

1. Sending a message
2. Receiving a message
3. Updating values
4. Synchronizing with other processes
5. ...

- Visible Actions
- Internal Actions

Internal and External Actions

An elementary action of a system represents the atomic (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the other.

Actions represent various activities of concurrent systems:

1. Sending a message
2. Receiving a message
3. Updating values
4. Synchronizing with other processes
5. ...

- Visible Actions
- Internal Actions

Internal and External Actions

An elementary action of a system represents the atomic (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the other.

Actions represent various activities of concurrent systems:

1. Sending a message
2. Receiving a message
3. Updating values
4. Synchronizing with other processes
5. ...

- Visible Actions
- Internal Actions

Internal and External Actions

An elementary action of a system represents the atomic (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the other.

Actions represent various activities of concurrent systems:

1. Sending a message
2. Receiving a message
3. Updating values
4. Synchronizing with other processes
5. ...

- Visible Actions
- Internal Actions

Internal and External Actions

An elementary action of a system represents the atomic (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the other.

Actions represent various activities of concurrent systems:

1. Sending a message
2. Receiving a message
3. Updating values
4. Synchronizing with other processes
5. ...

- Visible Actions
- Internal Actions

Internal and External Actions

An elementary action of a system represents the atomic (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the other.

Actions represent various activities of concurrent systems:

1. Sending a message
2. Receiving a message
3. Updating values
4. Synchronizing with other processes
5. ...

We have two main types of atomic actions:

- Visible Actions
- Internal Actions

Internal and External Actions

An elementary action of a system represents the atomic (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the other.

Actions represent various activities of concurrent systems:

1. Sending a message
2. Receiving a message
3. Updating values
4. Synchronizing with other processes
5. ...

We have two main types of atomic actions:

- Visible Actions
- Internal Actions

Internal and External Actions

An elementary action of a system represents the atomic (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the other.

Actions represent various activities of concurrent systems:

1. Sending a message
2. Receiving a message
3. Updating values
4. Synchronizing with other processes
5. ...

We have two main types of atomic actions:

- Visible Actions
- Internal Actions

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?

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

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?

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

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?

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

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?

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

As XML?

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

Why operators for describing systems - ctd

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. $p|q$)
- to model systems interaction
- to abstract from details
- to represent infinite systems

Why operators for describing systems - ctd

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. $p|q$)
- to model systems interaction
- to abstract from details
- to represent infinite systems

Why operators for describing systems - ctd

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. $p|q$)
- to model systems interaction
- to abstract from details
- to represent infinite systems

Operational Semantics

To each process built using the above operators we associate an LTS by relying on structural induction to define the meaning of each operator.

Definition (Inference Systems)

An inference system is a set of inference rules of the form

$$\frac{p_1, \cdots, p_n}{q}$$

In our case for a generic operator *op* we shall have one or more rules like:

$$\frac{E_{i_1} \xrightarrow{\alpha_1} E'_{i_1} \quad \cdots \quad E_{i_m} \xrightarrow{\alpha_m} E'_{i_m}}{op(E_1, \cdots, E_n) \xrightarrow{\alpha} op(E'_1, \cdots, E'_n)} \qquad \text{where } \{i_1, \cdots, i_m\} \subseteq \{1, \cdots, n\}.$$

Note that the states of the LTS are named with processes/expressions

The Elegance of Operational Semantics

Automata as terms

Few SOS rules define all the automata that can ever be specified with the chosen operators. Given any term, the rules are used to derive the corresponding automaton. The set of rules is fixed once and for all.

Structural induction

The interaction of complex systems is defined in terms of the behavior of their components.

A remark

The LTS is the least one satisfying the inference rules.

Rule induction

A property is true for the whole LTS if whenever it holds for the premises of each rule, it holds also for the conclusion.

The Elegance of Operational Semantics

Automata as terms

Few SOS rules define all the automata that can ever be specified with the chosen operators. Given any term, the rules are used to derive the corresponding automaton. The set of rules is fixed once and for all.

Structural induction

The interaction of complex systems is defined in terms of the behavior of their components.

A remark

The LTS is the least one satisfying the inference rules.

Rule induction

A property is true for the whole LTS if whenever it holds for the premises of each rule, it holds also for the conclusion.

The Elegance of Operational Semantics

Automata as terms

Few SOS rules define all the automata that can ever be specified with the chosen operators. Given any term, the rules are used to derive the corresponding automaton. The set of rules is fixed once and for all.

Structural induction

The interaction of complex systems is defined in terms of the behavior of their components.

A remark

The LTS is the least one satisfying the inference rules.

Rule induction

A property is true for the whole LTS if whenever it holds for the premises of each rule, it holds also for the conclusion.

The Elegance of Operational Semantics

Automata as terms

Few SOS rules define all the automata that can ever be specified with the chosen operators. Given any term, the rules are used to derive the corresponding automaton. The set of rules is fixed once and for all.

Structural induction

The interaction of complex systems is defined in terms of the behavior of their components.

A remark

The LTS is the least one satisfying the inference rules.

Rule induction

A property is true for the whole LTS if whenever it holds for the premises of each rule, it holds also for the conclusion.

Presentations of Labelled Transition Systems

Process Algebra as denotations of LTS

- LTS are represented by terms of process algebras.
- Terms are interpreted via operational semantics as LTS.

Process Algebra Basic Principles

1. Define a few elementary (atomic) processes modelling the simplest process behaviour;
2. Define appropriate composition operations to build more complex process behaviour from (existing) simpler ones.

Regular Expressions as Process Algebras

Syntax of Regular Expressions

$E \ ::= \ 0 \mid 1 \mid a \mid E+E \mid E \cdot E \mid E^* \ \textit{with}\ a \in A \ \textit{and} \ -\textit{below}- \ \mu \in A \cup \{\varepsilon\}$

Operational Semantics of Regular Expressions

(Tic)	$\dfrac{}{1 \xrightarrow{\varepsilon} 1}$	(Atom)	$\dfrac{}{a \xrightarrow{a} 1}$
(Sum$_1$)	$\dfrac{e \xrightarrow{\mu} e'}{e+f \xrightarrow{\mu} e'}$	(Sum$_2$)	$\dfrac{f \xrightarrow{\mu} f'}{e+f \xrightarrow{\mu} f'}$
(Seq$_1$)	$\dfrac{e \xrightarrow{\mu} e'}{e \cdot f \xrightarrow{\mu} e' \cdot f}$	(Seq$_2$)	$\dfrac{e \xrightarrow{\varepsilon} 1}{e \cdot f \xrightarrow{\varepsilon} f}$
(Star$_1$)	$\dfrac{}{e^* \xrightarrow{\varepsilon} 1}$	(Star$_2$)	$\dfrac{e \xrightarrow{\mu} e'}{e^* \xrightarrow{\mu} e' \cdot e^*}$

Table: We assume $a \in A$ and $\mu \in A \cup \{\varepsilon\}$.