Formalisation of component based systems

Lars-Åke Fredlund

Today’s Lecture

- Explore methods for component specification

Today’s Lecture

- Explore methods for component specification
- Methods will cover at least **specification of input/output types** and partial descriptions of **functional behaviour**

Today’s Lecture

- Explore methods for component specification
- Methods will cover at least **specification of input/output types** and partial descriptions of **functional behaviour**
- We will mention extensions to specify concurrent behaviour, timing behaviour, and so on (check transparencies for more details)

Today's Lecture

- Explore methods for component specification
- Methods will cover at least **specification of input/output types** and partial descriptions of **functional behaviour**
- We will mention extensions to specify concurrent behaviour, timing behaviour, and so on (check transparencies for more details)
- Methods will be based on formal methods

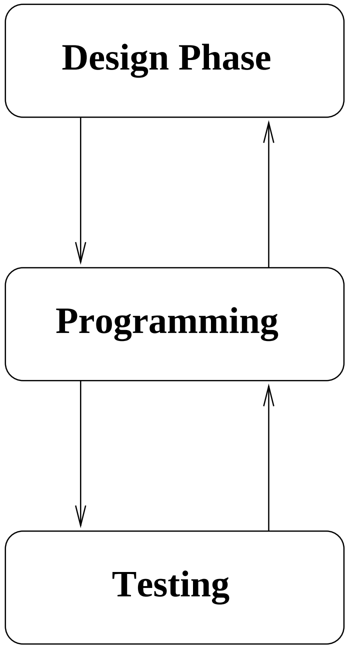
Formalisations

What do we mean by formalisations?

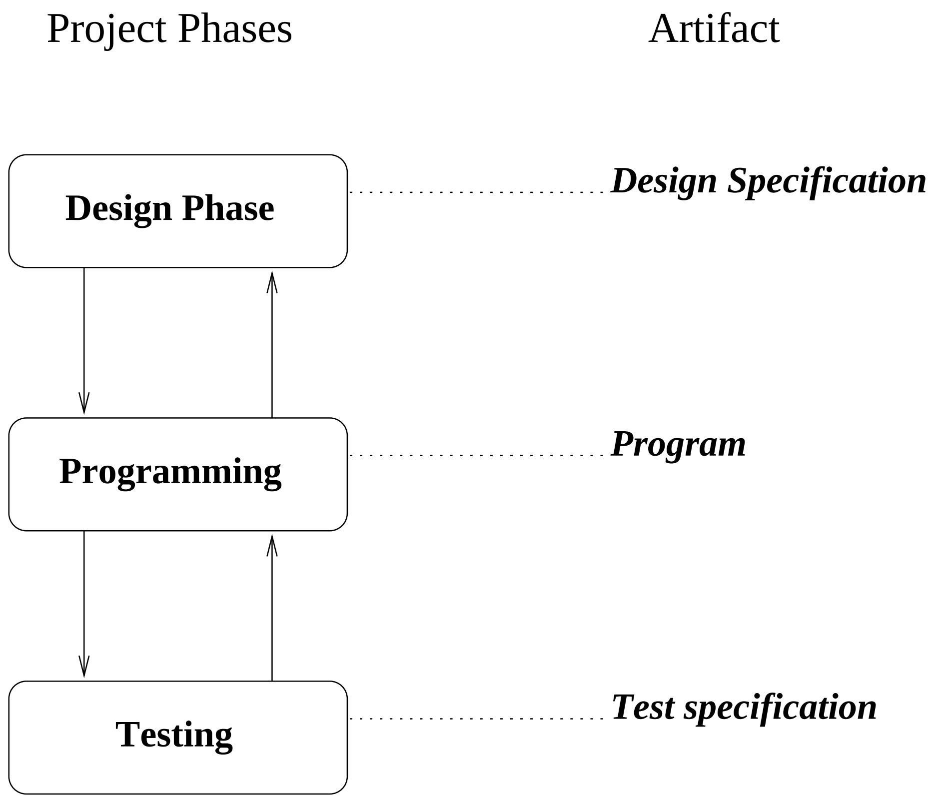
- Components (or systems) specified using *formal methods*
- Formal Methods are based in *sound mathematics* and provide the ability to *reason* rigourously about programs and specifications
- Reasoning may be automatic or manual

Formal Methods in a Project Design Cycle

Project Phases



Formal Methods in a Project Design Cycle



Formal Methods in a Project Design Cycle

Project Phases	Artifact	Formal or informal
Design Phase	***Design Specification***	*Normally informal, but formalizable*
Programming	***Program***	*Already formal!*
Testing	***Test specification***	*A set of tests, normally a mix of formal and informal*

Example: formalized Test Specifications

Since formal specifications have a precise meaning, we can use them for **analysis**

- Test Specification: a formalization of which tests to run, using a formal language

Facultad de Informática
Universidad Politécnica de Madrid

Example: formalized Test Specifications

Since formal specifications have a precise meaning, we can use them for **analysis**

- Test Specification: a formalization of which tests to run, using a formal language
- Analysis possible – measures of how **good** the testing process is – how big part of the program covered by tests:
  - how large percentage of the *lines* of the program are tested
  - **or**: how large percentage of the *paths* through the program tested
  - **or**: how many *states* of the program covered by tests

Facultad de Informática
Universidad Politécnica de Madrid

Model Driven Engineering

- We construct *models* of systems

Model Driven Engineering

- We construct *models* of systems
- A model *abstracts* from aspects of the real system

Model Driven Engineering

- We construct *models* of systems
- A model *abstracts* from aspects of the real system
- But can be used to *predict* some properties of the real system (e.g., for testing, . . .)

Model Driven Engineering

- We construct *models* of systems
- A model *abstracts* from aspects of the real system
- But can be used to *predict* some properties of the real system (e.g., for testing, . . .)
- Problem: how to keep the model “synchronised” with the system under development

Model Driven Engineering

- We construct *models* of systems
- A model *abstracts* from aspects of the real system
- But can be used to *predict* some properties of the real system (e.g., for testing, . . .)
- Problem: how to keep the model “synchronised” with the system under development
- Many initiatives: (Model-drive architecture (MDA) from OMG)

Non-software models

Building models is standard practice in “normal engineering”:

Non-software models

Building models is standard practice in “normal engineering”:

A real airplane (Saab Safir):

Facultad de Informática
Universidad Politécnica de Madrid

Non-software models

Building models is standard practice in “normal engineering”:

A real airplane (Saab Safir):

A 1:72 scale plastic *model* kit:

assembled by an experienced *tester*:

Benefits of formalized Design Specification

- Suppose we have a formalized design specification **(a system model)**
- We can then **derive** system tests from the *formal design specification*
- Test metrics become:
  - how big part of the *design specification* tested
  - That is, which percentage of paths through the *design specification* tested against the program, and which percentage of *design specification* states have been tested
- Other uses:
  - Generating a program (implementation) from the design specification
  - Validating/simulating high-level specifications: do they make sense?

Why Study Formalisation (and analysis)?

Formal methods have a bad reputation, but are becoming more and more used

- To document design decisions (UML)
- Concise specifications of tests (QuickCheck)
- In hardware (to verify Intel CPUs)
- In software:
    - Microsoft, Linux: security analysis, device driver safety, . . .
    - Static analysis in compilers to detect runtime errors at compile time
- Often works well in limited domain settings (with special rules for how to write software)
- In general very useful techniques for verification of safety critical systems or in situations where failures are very costly
- **Important**: formal methods are *debugging techniques*. They cannot guarantee correctness, but can make errors less likely

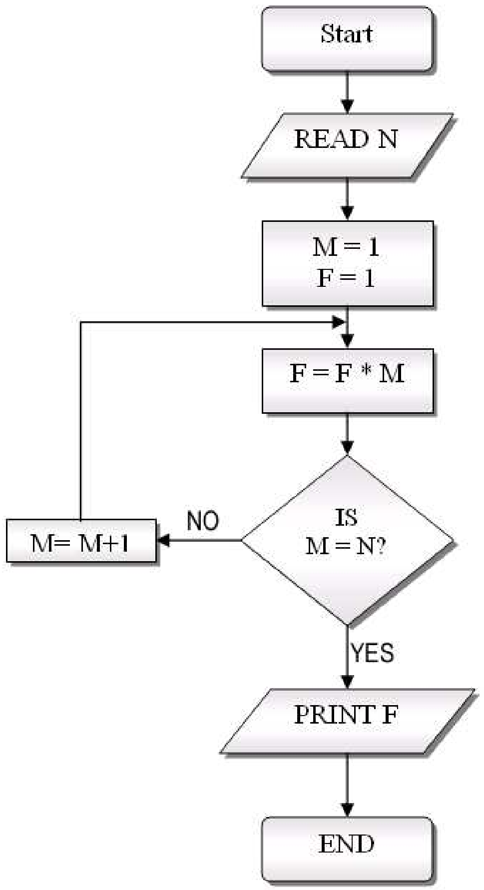
Formalisation

- In general formalisation is a big area – we will only introduce the topic here
- Today: *formalisation* examples,

  following lecture: *analysis* examples

Formalisations& Verification – History

- Turing verified programs
- Floyd: *program flowcharts*



Central Concepts

- Pre– and post–condition

  If a pre–condition *pre* holds before a statement, the post–condition *post* holds after

Central Concepts

- Pre– and post–condition

  If a pre–condition *pre* holds before a statement, the post–condition *post* holds after

- Invariant

  A property which holds always during an execution
  Loop invariants are used to characterise loop behaviour

Central Concepts

- Pre– and post–condition

  If a pre–condition *pre* holds before a statement, the post–condition *post* holds after

- Invariant

  A property which holds always during an execution
  Loop invariants are used to characterise loop behaviour

- Termination condition

  Specifies under which condition a computation terminates

  Usually proved by providing a *measure* – something which decreases during a computation, but cannot go on decreasing forever

Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:

$$\{Pre\}\ \ Command\ \ \{Post\}$$

Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:

$$\{Pre\}\ \ Command\ \ \{Post\}$$

- Example proof rules:

$$\frac{\{c \wedge I\}\ \ body\ \ \{I\}}{\{I\ \}\ \ while\ c\ do\ body\ \ \{\neg c \wedge I\}} \qquad \frac{\phi' \supset \phi \qquad \psi \supset \psi' \qquad \{\phi\}\ c\ \{\psi\}}{\{\phi'\}\ c\ \{\psi'\}}$$

Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:

$$\{Pre\}\ \ Command\ \ \{Post\}$$

- Example proof rules:

$$\frac{\{C \land I\}\ \ body\ \ \{I\}}{\{I\ \}\ \ while\ C\ do\ body\ \ \{\neg C \land I\}} \qquad \frac{\phi' \supset \phi \qquad \psi \supset \psi' \qquad \{\phi\}\ C\ \{\psi\}}{\{\phi'\}\ C\ \{\psi'\}}$$

- An example proof:

Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:

$$\{Pre\}\ \ Command\ \ \{Post\}$$

- Example proof rules:

$$\frac{\{C \wedge I\}\ \ body\ \ \{I\}}{\{I\ \}\ \ while\ C\ do\ body\ \ \{\neg C \wedge I\}} \qquad \frac{\phi' \supset \phi \qquad \psi \supset \psi' \qquad \{\phi\}\ C\ \{\psi\}}{\{\phi'\}\ C\ \{\psi'\}}$$

- An example proof:

$$\{i = N \wedge j = 0\}\ \texttt{while i>0 do i:=i-1;j:=j+1}\ \{j = N\}$$

Facultad de Informática
Universidad Politécnica de Madrid

Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:

$$\{Pre\}\ \ Command\ \ \{Post\}$$

- Example proof rules:

$$\frac{\{C \wedge I\}\ \ body\ \ \{I\}}{\{I\ \}\ \ while\ C\ do\ body\ \ \{\neg C \wedge I\}} \qquad \frac{\phi' \supset \phi \qquad \psi \supset \psi' \qquad \{\phi\}\ C\ \{\psi\}}{\{\phi'\}\ C\ \{\psi'\}}$$

- An example proof:

$$\frac{\frac{}{\{j = N - i\}\texttt{while i>0 do i:=i-1;j:=j+1}\ \{i = 0 \wedge j = N - i\}} \qquad i = N \wedge j = 0 \supset j = N - i \qquad i = 0 \wedge j = N - i \supset j = N}{\{i = N \wedge j = 0\}\ \texttt{while i>0 do i:=i-1;j:=j+1}\ \{j = N\}}$$

Facultad de Informática
Universidad Politécnica de Madrid

Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:

$$\{Pre\}\ \ Command\ \ \{Post\}$$

- Example proof rules:

$$\frac{\{C \wedge I\}\ \ body\ \ \{I\}}{\{I\ \}\ \ while\ C\ do\ body\ \ \{\neg C \wedge I\}} \qquad \frac{\phi' \supset \phi \qquad \psi \supset \psi' \qquad \{\phi\}\ C\ \{\psi\}}{\{\phi'\}\ C\ \{\psi'\}}$$

- An example proof:

$$\frac{\dfrac{\overline{\{i > 0 \wedge j = N - i\}\ i := i - 1;\ j := j + 1\ \{j = N - i\}}}{\{j = N - i\}\texttt{while i>0 do i:=i-1;j:=j+1}\ \{i = 0 \wedge j = N - i\}} \qquad i = N \wedge j = 0 \supset j = N - i \qquad i = 0 \wedge j = N - i \supset j = N}{\{i = N \wedge j = 0\}\ \texttt{while i>0 do i:=i-1;j:=j+1}\ \{j = N\}}$$

Facultad de Informática
Universidad Politécnica de Madrid

Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:

$$\{Pre\}\ \ Command\ \ \{Post\}$$

- Example proof rules:

$$\frac{\{C \wedge I\}\ \ body\ \ \{I\}}{\{I\ \}\ \ while\ C\ do\ body\ \ \{\neg C \wedge I\}} \qquad \frac{\phi' \supset \phi \qquad \psi \supset \psi' \qquad \{\phi\}\ C\ \{\psi\}}{\{\phi'\}\ C\ \{\psi'\}}$$

- An example proof:

$$\frac{\dfrac{\dfrac{\models (N-i)+1 = N-(i-1)}{\{i>0 \wedge j=N-i\}\ i:=i-1;\ j:=j+1\ \{j=N-i\}}}{\{j=N-i\}\texttt{while i>0 do i:=i-1;j:=j+1}\ \{i=0 \wedge j=N-i\}} \qquad i=N \wedge j=0 \supset j=N-i \qquad i=0 \wedge j=N-i \supset j=N}{\{i=N \wedge j=0\}\ \texttt{while i>0 do i:=i-1;j:=j+1}\ \{j=N\}}$$

Formalisations – History

- Dijkstra: weakest pre condition: $wp(c)$
- Owicki and Gries: extensions to concurrency
- Lamport: proving an invariant I:
  - I holds in the initial state of the program
  - For all program statements s prove $\{I\}\ s\ \{I\}$
    (if I holds before the statement s, it should afterwards also)

Correctness claims – classical

For classical terminating programs (or functions) a number of properties needs to be proved

- *Partial correctness*: if the program halts it satisfies a property
- *Termination*: the program halts
- *Total correctness*: both Partial correctness and Termination

Correctness claims – reactive systems

- The previous correctness claims are sufficient for *terminating* programs
- But *reactive systems* keep on running
- A reactive systems is a system that responds to stimuli (input) and responds with actions (output) – **a process**
- To formulate correctness properties about reactive systems people started experimenting with *temporal logics*, *program equivalences*, and so on. . .

Temporal logic

- Pneuli 1977: added discrete and linear time operators to propositional logic, to be able to specify properties of reactive systems
- Program meaning (semantics):
  - a *program state* s maps the program variables to values
  - a *run* of the program is an infinite sequence of program states $(s_0, s_1, s_2, \ldots)$ from an initial state s_0
  - for a terminating system simply add a self-loop in the terminating state to yield an infinite run
  - the *semantics* of a program p is its set of runs, $\| p \|$
  - If the program is **nondeterministic** (or accepts input) there will be more than one run of the program

Runs of concurrent programs: examples

Consider the following simple shared variable program:

```
if x>0 then x:=x-1 || if x<3 then x:=x+1
```

where `S1 || S2` runs the atomic statements `S1` and `S2` in parallel

Facultad de Informática Universidad Politécnica de Madrid

Runs of concurrent programs: examples

Consider the following simple shared variable program:

```
if x>0 then x:=x-1 || if x<3 then x:=x+1
```

where `S1 || S2` runs the atomic statements `S1` and `S2` in parallel

Its runs starting from the state `x=0` is the **infinite** set:

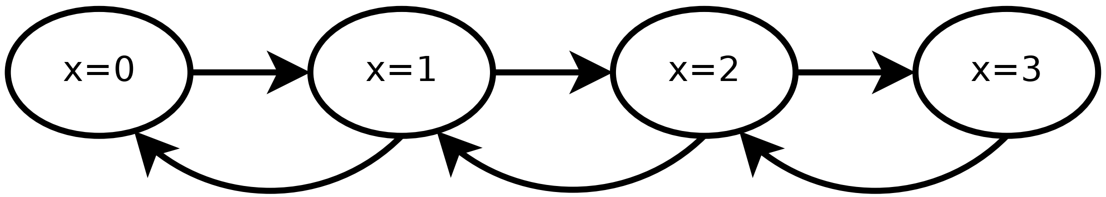
$$\left\{\begin{array}{l} \langle x=0\rangle\cdot\langle x=1\rangle\cdot\langle x=0\rangle\cdot\ldots \\ \langle x=0\rangle\cdot\langle x=1\rangle\cdot\langle x=2\rangle\cdot\langle x=1\rangle\cdot\ldots \\ \langle x=0\rangle\cdot\langle x=1\rangle\cdot\langle x=2\rangle\cdot\langle x=3\rangle\cdot\langle x=2\rangle\ldots \\ \ldots \end{array}\right\}$$

Program runs

We can also depict the runs

$$\left\{ \begin{array}{l} \langle x = 0\rangle \cdot \langle x = 1\rangle \cdot \langle x = 0\rangle \cdot \ldots \\ \langle x = 0\rangle \cdot \langle x = 1\rangle \cdot \langle x = 2\rangle \cdot \langle x = 1\rangle \cdot \ldots \\ \langle x = 0\rangle \cdot \langle x = 1\rangle \cdot \langle x = 2\rangle \cdot \langle x = 3\rangle \cdot \langle x = 2\rangle \ldots \\ \ldots \end{array} \right\}$$

as a state graph:



Atomicity

Consider the parallel program

```
if x>0 then x:=x-1 || if x>0 then x:=x-1
```

with the starting state $\langle x = 3 \rangle$

Atomicity

Consider the parallel program

```
if x>0 then x:=x-1 || if x>0 then x:=x-1
```

with the starting state $\langle x = 3 \rangle$

- If statements are **atomic** the program has the single run
  $\langle x = 3 \rangle \langle x = 2 \rangle \langle x = 1 \rangle \langle x = 0 \rangle$

Atomicity

Consider the parallel program

```
if x>0 then x:=x-1 || if x>0 then x:=x-1
```

with the starting state $\langle x = 3 \rangle$

- If statements are **atomic** the program has the single run $\langle x = 3 \rangle \ \langle x = 2 \rangle \ \langle x = 1 \rangle \ \langle x = 0 \rangle$
- If statements are **not atomic** the program has the two runs

$$\left\{ \begin{array}{l} \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \\ \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \cdot \langle x = -1 \rangle \end{array} \right\}$$

Atomicity

Consider the parallel program

```
if x>0 then x:=x-1 || if x>0 then x:=x-1
```

with the starting state $\langle x = 3 \rangle$

- If statements are **atomic** the program has the single run $\langle x = 3 \rangle\ \langle x = 2 \rangle\ \langle x = 1 \rangle\ \langle x = 0 \rangle$
- If statements are **not atomic** the program has the two runs

$$\left\{ \begin{array}{l} \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \\ \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \cdot \langle x = -1 \rangle \end{array} \right\}$$

- **Conclusion:** programming concurrent programs is *hard*

Atomicity

Consider the parallel program

```
if x>0 then x:=x-1 || if x>0 then x:=x-1
```

with the starting state $\langle x = 3 \rangle$

- If statements are **atomic** the program has the single run $\langle x = 3 \rangle \; \langle x = 2 \rangle \; \langle x = 1 \rangle \; \langle x = 0 \rangle$
- If statements are **not atomic** the program has the two runs

$$\left\{ \begin{array}{l} \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \\ \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \cdot \langle x = -1 \rangle \end{array} \right\}$$

- **Conclusion:** programming concurrent programs is *hard*
- The Erlang programming language we shall hear about soon has **good** concurrency primitives

Facultad de Informática Universidad Politécnica de Madrid

Temporal logic – operators

Classical linear temporal operators (defined over runs):

- *Always* ϕ

  ϕ holds in all future states of the run
- *Eventually* ϕ

  ϕ holds in some future state of the run
- *Next* ϕ

  ϕ holds in the next state
- ϕ_1 *Until* ϕ_2

  ϕ_1 holds in all states until ϕ_2 holds
- And the normal ones: negation $\neg\phi$, conjunction $\phi_1 \wedge \phi_2$, implication $\phi_1 \supset \phi_2, \ldots$
- And propositional operators (over variables): $x < y$, $even(z)$, $\ldots$

Facultad de Informática
Universidad Politécnica de Madrid

Temporal logic – meaning

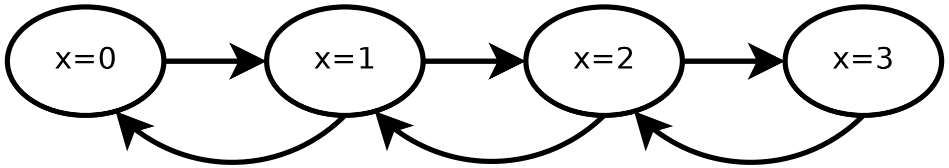
- A program p satisfies a formula ϕ when all the runs of the program are satisfied by the formula
- The logic is linear because it doesn't talk about the branching structure of the state graph of the program (*what is set of the possible next states of the program*)
- So called *branching time* logics do consider the branching structure of the state graph of the program

Temporal logic – examples

Consider the atomic parallel program

```
if x>0 then x:=x-1 || if x<3 then x:=x+1
```

with the starting state $\langle x = 3 \rangle$ and the state graph

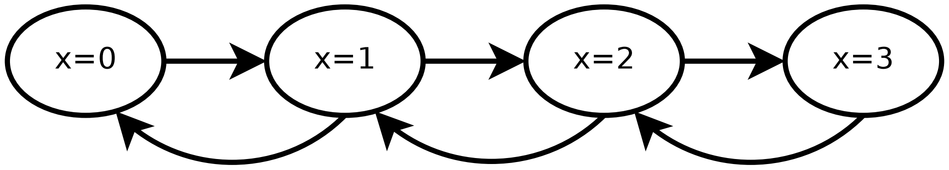


Temporal logic – examples

Consider the atomic parallel program

```
if x>0 then x:=x-1 || if x<3 then x:=x+1
```

with the starting state $\langle x = 3 \rangle$ and the state graph



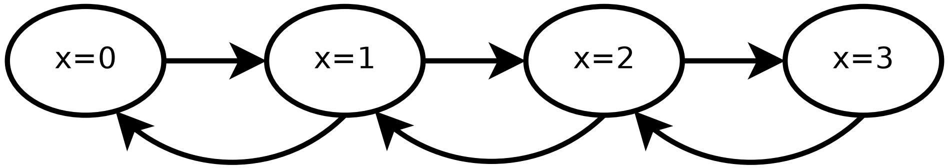
- Does *Always* $x \geq 0$ hold?

Temporal logic – examples

Consider the atomic parallel program

```
if x>0 then x:=x-1 || if x<3 then x:=x+1
```

with the starting state $\langle x = 3 \rangle$ and the state graph



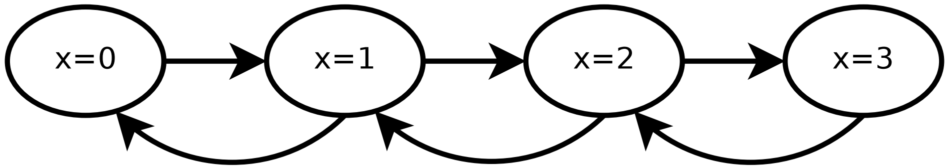
- Does *Always* $x \geq 0$ hold?
- **Yes**; if x=0 then the guard prevents further decrease

Temporal logic – examples

Consider the atomic parallel program

```
if x>0 then x:=x-1 || if x<3 then x:=x+1
```

with the starting state $\langle x = 3 \rangle$ and the state graph



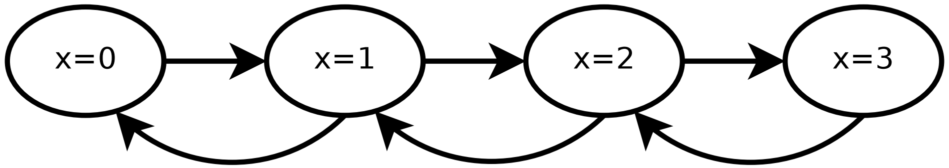
- Does *Always* $x \geq 0$ hold?
- **Yes**; if x=0 then the guard prevents further decrease
- Does *Always* $(x = 3 \supset \textit{Eventually}\ x = 0)$ hold?

Temporal logic – examples

Consider the atomic parallel program

```
if x>0 then x:=x-1 || if x<3 then x:=x+1
```

with the starting state $\langle x = 3 \rangle$ and the state graph



- Does *Always* $x \geq 0$ hold?
- **Yes**; if x=0 then the guard prevents further decrease
- Does *Always* $(x = 3 \supset$ *Eventually* $x = 0)$ hold?
- **No**; there is a run $\langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \ldots$

General temporal logic patterns

- *Eventually* $\phi \equiv \neg$ *Always* $(\neg \phi)$

Facultad de Informática
Universidad Politécnica de Madrid

General temporal logic patterns

- *Eventually* $\phi \equiv \neg$ *Always* $(\neg \phi)$
- A *safety property* expresses that *something bad* – ϕ – *never happens*: *Always* $\neg \phi$

General temporal logic patterns

- *Eventually* $\phi \equiv \neg$ *Always* $(\neg \phi)$
- A *safety property* expresses that *something bad* – ϕ – *never happens*: *Always* $\neg \phi$
- A *liveness property* expresses that something good – ϕ – eventually happens: *Eventually* ϕ

General temporal logic patterns

- *Eventually* $\phi \;\equiv\; \neg$ *Always* $(\neg\,\phi)$
- A *safety property* expresses that *something bad* – ϕ – *never happens*: *Always* $\neg\,\phi$
- A *liveness property* expresses that something good – ϕ – eventually happens: *Eventually* ϕ
- Often one uses *fairness assumptions* to rule out bad program behaviours

  ϕ eventually holds under the assumption that ψ doesn't always hold:

$$(\neg \textit{Always}\ \psi) \;\supset\; (\textit{Eventually}\ \phi)$$

Facultad de Informática
Universidad Politécnica de Madrid

Temporal logic

- Many variants exists:
  - Continuous time, time interval models
  - Additional operators (e.g., talking about past time)
  - Branching time logics: $\mathcal{A}\,\phi$ – ϕ holds on all program states that are reachable from the current one
- In general temporal logics are good for expressing program correctness properties
- But it is difficult to give *complete* descriptions of what a correct program behaviour is
- More on algorithms and tools for checking programs against properties later on. . .

Specification Languages

- A *specification* is an abstract definition of what the correct behaviour of a program is
- A specification abstracts away from irrelevant details
- Verification techniques (later) permits to check whether a program satisfies its specification
- Two large families of specification languages:
  - abstract *model* based (Z, VDM, Slam-SL)
  - and *algebraic* specifications (CCS, π calculus)

An model based language: Z

- Based on mathematical logic (first-order predicate logic) and set theory
- Objects are typed
- *Schemas* permits to specify in a modular fashion both static (data) and dynamic (behaviour) properties of a system
- The acceptable data states of a program are specified using predefined mathematical concepts (sequences, sets, . . .)
- Operations upon states are characterised using pre– and post–conditions

The Z specification language

Different Schema Types:

- State schemas characterises reachable states of a system using a state invariant; defines what must be respected by operations
- Operation schemas describe how operations with input parameters cause state changes, and return parameters
- Operation schemas are given using pre– and post–conditions

Z schema example: static part

$$Result \;::=\; ok \;|\; nospace$$

Queue

$$queue : seq\ \mathbb{N}$$

$$\#queue < 10$$

Init

$$Queue$$

$$queue = \langle\rangle$$

Z schema example, dynamic part

InsertOk

$\Delta Queue$
$insert? : \mathbb{N}$
$result! : Result$

$\#queue < 10$
$queue' = queue \frown \langle insert? \rangle$
$result! = ok$

InsertWithError

$\Xi Queue$
$insert? : \mathbb{N}$
$result! : Result$

$\#queue \geq 10$
$result! = nospace$

$$Insert = InsertOk \lor InsertWithError$$

Abstract models – Z

Proof challenges (for a theorem prover):

- An initial state exists
- The pre-condition of each operation guarantees that the resulting state exists (is a proper state). Consider:

BadOp

$$\Delta Queue$$
$$insert? : \mathbb{N}$$
$$result! : Result$$

$$queue' = queue \frown \langle insert? \rangle$$
$$result! = ok$$

Violates the state invariant

Facultad de Informática
Universidad Politécnica de Madrid

Z limitations and extensions

- Serious limitations with respect to:
  - Lack of modularisation and Object Orientation
  - Specifying reactive systems, realtime, concurrency
- Has been used in quite a few industrial projects (in the UK)
- A mathematical clean notation, expressive, but rather abstract (can be difficult to implement)
- To implement a Z spec one can use program derivation and program refinement techniques (in Z itself)
- Object-oriented extensions exists: Object-Z and Z++
- Nowadays the B method receives more attention

Axiomatic Specifications

- A system is specified as a set of abstract data types
- Operations on data are characterised as axioms of equality
- Rewrite rules define transitions between system states (instances of the data types)
- Specifications are **executable**
- Examples: Maude/rewriting logic, OBJ, FOOPS

Maude

- Home page: `http://maude.cs.uiuc.edu/`
- Origin at Stanford, USA
  (Messeguer and others, big Spanish community)
- Rewriting of equations modulo commutativity, associativity and idem-potency
- Equations are evaluated nondeterministically, in parallel
- Permits specification of reactive (and concurrent) systems
- Reflexive language (Maude can be represented in Maude) and object-oriented (inheritance, polymorphism)
- Uses reflection to control which equations and rewrite rules to apply (rewriting strategies)

Maud example

```
fmod NatQueue is
  sorts NatQueue .
  protecting NAT .

  op empty : -> NatQueue [ctor] .
  op enq : Nat NatQueue -> NatQueue [ctor] .
  op head : NatQueue -> Nat .
  op deq : NatQueue -> NatQueue .

  vars N N1 : Nat . vars Q : NatQueue .

  eq head(enq(N,empty)) = N .
  eq head(enq(N,enq(N1,Q))) = head(enq(N1,Q)) .

  eq deq(enq(N,empty)) = empty .
  eq deq(enq(N,enq(N1,Q))) = enq(N,deq(enq(N1,Q))) .
endfm
```

Usage examples

- What is the head of the queue after inserting 3 and then 2?

```
red head(enc(2,enc(3,empty))) .
```

Usage examples

- What is the head of the queue after inserting 3 and then 2?

  `red head(enc(2,enc(3,empty))) .`

- Answer: `result NzNat:  3`

Maud example: with rewrite rules

Rewrite rules express transitions between states:

```
mod CommChannel is
  protecting NatQueue .

  vars N : Nat .
  vars Q : NatQueue .

  rl [receive] :
       enq(N,Q) => deq(enq(N,Q)) .

  rl [loose_msg] :
       enq(N,Q) => Q .

  rl [duplicate_msg] :
       enq(N,Q) => enq(N,enq(N,Q)) .
endm
```

Usage examples

- What are the possible queues resulting after at most two transitions from the state $2 \cdot 1$?

  `search [,2] enq(2,enq(1,empty)) =>+ q:NatQueue .`

- Answer: `11 states`:

$$
\begin{array}{lll}
\epsilon & & \\
2 & 1 & \\
2 \cdot 2 & 1 \cdot 1 & 2 \cdot 1 \\
2 \cdot 2 \cdot 1 & 2 \cdot 1 \cdot 1 & \\
2 \cdot 2 \cdot 2 \cdot 1 & 2 \cdot 2 \cdot 1 \cdot 1 & 2 \cdot 1 \cdot 1 \cdot 1
\end{array}
$$

- Path to $2 \cdot 1 \cdot 1 \cdot 1$:

```
Maude> show path 10 .
state 0, NatQueue: enq(2, enq(1, empty))
===[ rl enq(N, Q) => enq(N, enq(N, Q)) [label duplicate_msg] . ]===>
state 4, NatQueue: enq(2, enq(1, enq(1, empty)))
===[ rl enq(N, Q) => enq(N, enq(N, Q)) [label duplicate_msg] . ]===>
state 10, NatQueue: enq(2, enq(1, enq(1, enq(1, empty))))
```

Maude conclusions

- Useful for developing executable prototypes rapidly
- Yields reasonable efficient prototypes
- But specifications become pretty algorithmic (not abstract)

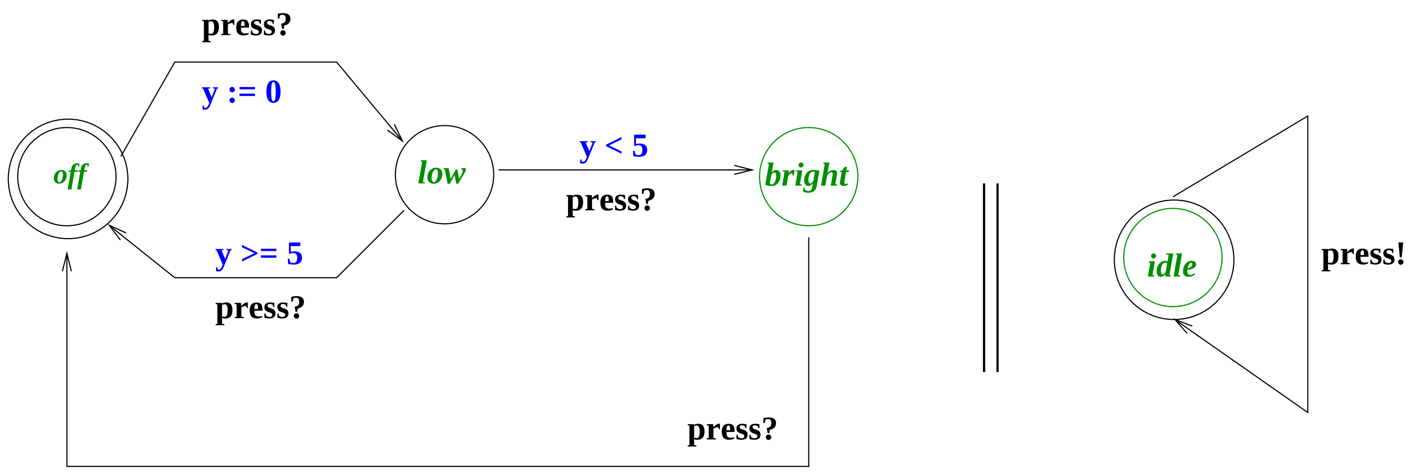
Specifying real-time and hybrid systems

- Specification languages generally tailored to the task of constructing verifyable models
- Typical system: network of clocked automata
- Languages and verification systems: UPPAAL (`http://www.uppaal.com`)
- Alternatives: modelling using real-time UML variants

Facultad de Informática
Universidad Politécnica de Madrid

A Lamp Example in UPPAAL

A lamp that has two intensities (*low* and *bright*), and a user:

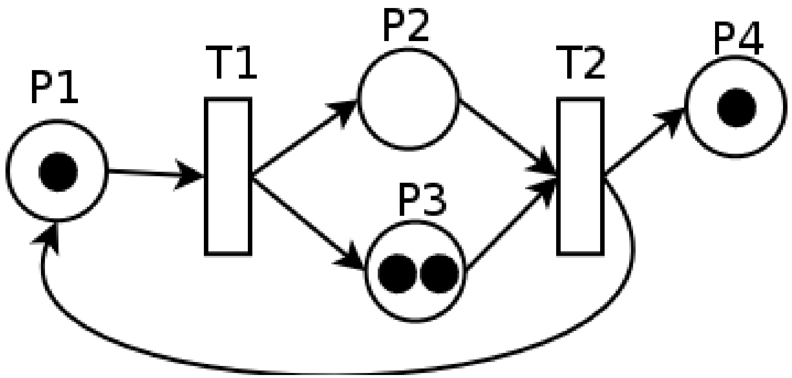


If the user presses the button (`press!`) twice within 5 time units the intensity of the lamp is set to bright

Facultad de Informática Universidad Politécnica de Madrid

Formalisation of Concurrency and Distribution

- *Petri-Nets*: a mainly graphical notation for transition systems:



  Nondeterministic, highly concurrent
  Intuitive with a good notation, but do they scale?

- State machines that communicate by exchanging messages: SDL, Promela, I/O-automatons, cleanly written Erlang, . . .

  Often serious specification languages useful for checking complex systems

- Process algebras: CCS, CSP, π-calculus, ambient calculus

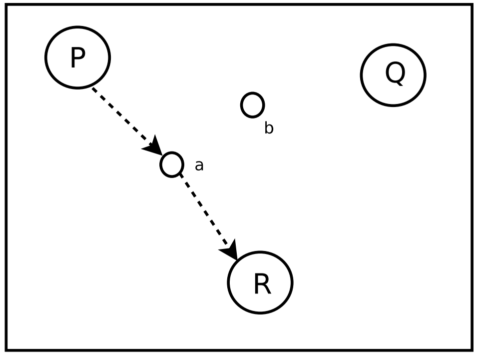
  Mathematically clean formalisms, often less suited for specifying larger systems

Process Algebras

- A large variant of calculi for providing a mathematically elegant theory of concurrency
  - For basic concerns: CCS, CSP
  - With extensions to mobility: π-calculus
  - With extensions to distribution: mobile ambients
  - With abstract data types: LOTOS, μCRL
  - With (many) extensions to real-time, to stochastic behaviours, to . . .
- As a summary: popular as a (mathematical) tool for reasoning about concurrency; for real programs?
- Due to their popularity one should have a basic knowledge of the field

Process Algebras: CCS

- CCS is a very basic process algebra (due to Milner 1980)
- Basic entities are **processes** and **ports** (used for binary communication between two processes)
- We let $P, Q, R, \ldots$ stand for processes and $a, b, c, \ldots$ for ports
- A process/port graph:



CCS syntax

- Communication by synchronisation: (a is a port)
  output action $\overline{a}(v)$
  input action $a(x)$
  or internal action τ

  If there is no value being sent or received we omit the action parameter: $\overline{a}(v)$ becomes $\overline{a}$ and $a(x)$ becomes a

- Sequential composition: $\alpha.P$
  where α is an action (input or output action, or internal τ)
  Behaviour: after performing a it behaves as P

- Choice: $P + Q$ can behave as P or as Q

- Parallel behaviour: the agent $P \mid Q$ behaves as P running in parallel with Q

- The agent which can do nothing: 0

A simple example: a coffee/tea machine

- A simple (one-use) coffee machine:

  $$\mathit{coin}.\left(\overline{\mathit{coffee}}.0 + \overline{\mathit{tea}}.0\right)$$

A simple example: a coffee/tea machine

- A simple (one-use) coffee machine:

  $\textit{coin}.\left(\overline{\textit{coffee}}.0 + \overline{\textit{tea}}.0\right)$

- A user that always wants tea:

  $\overline{\textit{coin}}.\textit{tea}.0$

A simple example: a coffee/tea machine

- A simple (one-use) coffee machine:

  $$\mathit{coin}.\left(\overline{\mathit{coffee}}.0 + \overline{\mathit{tea}}.0\right)$$

- A user that always wants tea:

  $$\overline{\mathit{coin}}.\mathit{tea}.0$$

- The combination of a user and the coffee machine:

  $$\mathit{coin}.\left(\overline{\mathit{coffee}}.0 + \overline{\mathit{tea}}.0\right) \;\mid\; \overline{\mathit{coin}}.\mathit{tea}.0$$

Process Algebras: CCS operators part II

- Restriction: the agent $P \setminus \{a\}$ cannot communicate with the environment using the port a
- Relabelling: in communications with its environment the agent $P[f]$ relabels all channel names using the relabelling function f
- Recursive agents can be defined: $P \stackrel{\text{def}}{=} a.(P \mid b.0)$
- And simple test on boolean conditions: *if* b *then* P *else* Q

A simple example: a coffee/tea machine continued

- A simple coffee machine:

$$\mathit{CM1} \stackrel{\text{def}}{=} \mathit{coin}.\left(\overline{\mathit{coffee}}.\mathit{CM1} + \overline{\mathit{tea}}.\mathit{CM1}\right)$$

- A user that always wants tea:

$$\mathit{User} \stackrel{\text{def}}{=} \overline{\mathit{coin}}.\mathit{tea}.0$$

- The combination of a user and the coffee machine:

$$(\mathit{User} \mid \mathit{CM1}) \setminus \{\mathit{coffee}, \mathit{tea}, \mathit{coin}\}$$

Defining the Behaviour of CCS Agents

- A transition rule based semantics, defined using the syntactic shape of terms, is often called a **structured operational semantics** (abbreviated **SOS**)
- Such semantics are used to define the meaning of many programming languages and systems
- One should at least have a basic grasp of how to read such semantic definitions
- We use an operational semantics to define the behaviour of CCS agents

CCS, Operational behaviour

Semantics defined by transition rules:

- prefix $\dfrac{}{\alpha.P \xrightarrow{\alpha} P}$

- choice $\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}, \quad \dfrac{P \xrightarrow{\alpha} P'}{Q + P \xrightarrow{\alpha} P'}$

- interleaving $\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}, \quad \dfrac{P \xrightarrow{\alpha} P'}{Q \mid P \xrightarrow{\alpha} Q \mid P'}$

- synchronisation$_l$ $\dfrac{P \xrightarrow{a(x)} P' \qquad Q \xrightarrow{\bar{a}(v)} Q'}{P \mid Q \xrightarrow{\tau} P'[v/x] \mid Q'}$

- synchronisation$_r$ $\dfrac{Q \xrightarrow{a(x)} Q' \qquad P \xrightarrow{\bar{a}(v)} P'}{P \mid Q \xrightarrow{\tau} P'[v/x] \mid Q'}$

CCS, Operational behaviour part II

- restriction $$\frac{P \xrightarrow{\alpha} P' \qquad a \notin fn(\alpha)}{P \setminus \{a\} \xrightarrow{\alpha} P' \setminus \{a\}}$$

  where $[fn(\alpha) \equiv \left\{ \begin{array}{cc} \{a\} & \text{if } \alpha = \bar{a}(v) \\ \{a\} & \text{if } \alpha = a(x) \\ \{\} & \text{if } \alpha = \tau \end{array} \right\}$

- relabelling $$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

- recursion $$\frac{P[v/x] \xrightarrow{\alpha} P' \qquad N(x) \stackrel{\text{def}}{=} P}{N(v) \xrightarrow{\alpha} P'}$$

- eval $$\frac{P \xrightarrow{\alpha} P' \qquad b\ true}{if\ b\ then\ P\ else\ Q \xrightarrow{\alpha} P'}, \quad \frac{P \xrightarrow{\alpha} P' \qquad b\ false}{if\ b\ then\ Q\ else\ P \xrightarrow{\alpha} P'}$$

CCS examples

- Coffee machine 1 always work:
  $\mathit{CM1} \stackrel{\mathrm{def}}{=} \mathit{coin}.\left(\overline{\mathit{coffee}}.\mathit{CM1} + \overline{\mathit{tea}}.\mathit{CM1}\right)$

CCS examples

- Coffee machine 1 always work:
  $CM1 \stackrel{\text{def}}{=} coin.\left(\overline{coffee}.CM1 + \overline{tea}.CM1\right)$

- Coffee machine 2 sometimes (unspecified when) only allows the coffee choice:
  $CM2 \stackrel{\text{def}}{=} coin.\left(\overline{coffee}.CM2 + \overline{tea}.CM2 + \tau.\overline{coffee}.CM2\right)$

CCS examples

- Coffee machine 1 always work:
  $\textit{CM1} \stackrel{\text{def}}{=} \textit{coin}.\left(\overline{\textit{coffee}}.\textit{CM1} + \overline{\textit{tea}}.\textit{CM1}\right)$

- Coffee machine 2 sometimes (unspecified when) only allows the coffee choice:
  $\textit{CM2} \stackrel{\text{def}}{=} \textit{coin}.\left(\overline{\textit{coffee}}.\textit{CM2} + \overline{\textit{tea}}.\textit{CM2} + \tau.\overline{\textit{coffee}}.\textit{CM2}\right)$

- A typical user that always wants tea:
  $\textit{User} \stackrel{\text{def}}{=} \overline{\textit{coin}}.\textit{tea}.0$

CCS examples

- Coffee machine 1 always work:
  $CM1 \stackrel{\text{def}}{=} coin.\left(\overline{coffee}.CM1 + \overline{tea}.CM1\right)$
- Coffee machine 2 sometimes (unspecified when) only allows the coffee choice:
  $CM2 \stackrel{\text{def}}{=} coin.\left(\overline{coffee}.CM2 + \overline{tea}.CM2 + \tau.\overline{coffee}.CM2\right)$
- A typical user that always wants tea:
  $User \stackrel{\text{def}}{=} \overline{coin}.tea.0$
- The combination of a user and machine 1 (let $S \equiv \{coffee, tea, coin\}$):
  $(User \mid CM1) \setminus S \xrightarrow{\tau}\xrightarrow{\tau} (0 \mid CM1) \setminus S$

Facultad de Informática
Universidad Politécnica de Madrid

CCS examples

- Coffee machine 1 always work:
  $$\mathit{CM1} \stackrel{\text{def}}{=} \mathit{coin}.\left(\overline{\mathit{coffee}}.\mathit{CM1} + \overline{\mathit{tea}}.\mathit{CM1}\right)$$
- Coffee machine 2 sometimes (unspecified when) only allows the coffee choice:
  $$\mathit{CM2} \stackrel{\text{def}}{=} \mathit{coin}.\left(\overline{\mathit{coffee}}.\mathit{CM2} + \overline{\mathit{tea}}.\mathit{CM2} + \tau.\overline{\mathit{coffee}}.\mathit{CM2}\right)$$
- A typical user that always wants tea:
  $$\mathit{User} \stackrel{\text{def}}{=} \overline{\mathit{coin}}.\mathit{tea}.0$$
- The combination of a user and machine 1 (let $s \equiv \{\mathit{coffee}, \mathit{tea}, \mathit{coin}\}$):
  $$(\mathit{User} \mid \mathit{CM1}) \setminus s \xrightarrow{\tau}\xrightarrow{\tau} (0 \mid \mathit{CM1}) \setminus s$$
- The combination of a user and machine 2 may deadlock after two machine steps:
  $$(\mathit{User} \mid \mathit{CM2}) \setminus s \xrightarrow{\tau}\xrightarrow{\tau} (\mathit{tea}.0 \mid \overline{\mathit{coffee}}.\mathit{CM2}) \setminus s$$

Specifications for Process Algebras

Suppose that we have written a complex agent P, and we want to develop a simpler specification for that agent. What can we do?

Two options:

- Write the specification as a temporal logic formula ϕ, and show that $P : \phi$ (P satisfies ϕ)
- Write the specification as another CCS agent S, and show that $P = S$, with regards to some notion of equality "$=$"

Process algebra – process equality

Crucial question: when do two processes P and Q exhibit the same behaviour?

- First question: what does it mean for P and Q to have the same behaviour?
- Do we require that they have (almost) the same set of traces? In practise this is often too weak, e.g., $CM1$ and (a slight variant of) $CM2$ have the same set of traces but have very different behaviour
- Or do we need a stronger notion of equivalence? There are many options out there: strong equivalence, observation equivalence, . . .

Proving processes equal

- Most algebras have an axiomatic theory, e.g., a set of equations of the type $P + Q \ = \ Q + P$, $P \mid 0 \ = \ P$ and so on...
- Hence two processes P and Q are equal if we can prove $P = Q$ using the axioms
- A more behavioural alternative is to find a *bisimulation relation* relating P and Q
- Two process P and Q are (strong bisimulation) equivalent if we can find a bisimulation relation S containing the pair (P, Q)

  A pair $(P, Q) \in S$ if and only if

  - If $P \xrightarrow{\alpha} P'$ for some α and P' then there exists a Q' such that $Q \xrightarrow{\alpha} Q'$, and $(P', Q') \in S$
  - If $Q \xrightarrow{\alpha} Q'$ for some α and Q' then there exists a P' such that $P \xrightarrow{\alpha} P'$, and $(P', Q') \in S$

  Often it is far easier to find a bisimulation relation than to use equational reasoning

π calculus

- CCS is a fairly static calculus – what if we allow names (channels) to be communicated?
- The result is the π calculus (Milner, Walker and Parrow – 1989)
- A process that receives a new name can later communicate using it (new communication capabilities arise during the execution)
- The distinction between channels and data is removed
- A very basic calculus (but expressive!) for experimenting with one form of mobility
- Nowadays very popular – inspiration for some standards proposals for composition languages of web services

π calculus operators

- Most operators come from CCS: 0 – the inactive process, choice $P + Q$, parallelism $P \mid Q$
- Communication primitives are different:
  - output: $\bar{x}\,y.P$: the name y is sent over the name x; then behaves as P
  - input: $x(w).P$: a name y is received on the channel x; then behaves as $P[y/w]$ (P with y substituted for w)
- Matching: $[x = y]P$ behaves as P if x is the same name as y, otherwise as 0
- Private names: $(x)P$ creates a new name x that is private to P
- Replication: $!P$ is equivalent to $!P \mid P$ (an infinite number of copies of P in parallel)

π calculus: name mobility

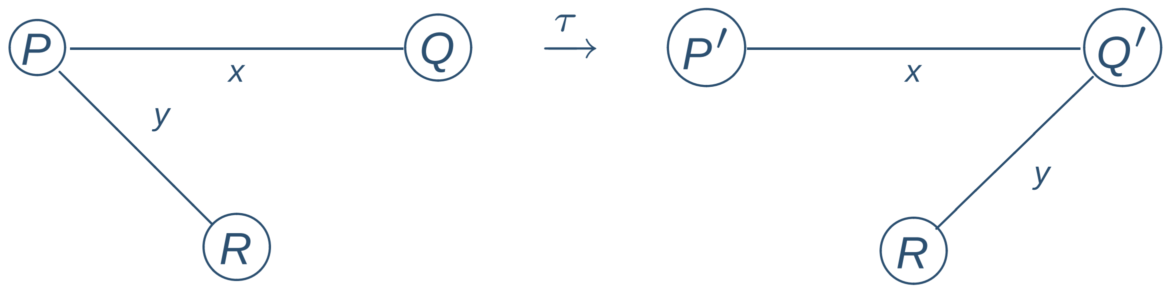
- Mobility example:

  Receive a new name at a and use the new name to send z:
  $a(x).\,\overline{x}\,z.\,P$

- Evolution of communication capabilities, let:
  $P(x,y) \stackrel{\text{def}}{=} \overline{x}\,y.P'(x)$ and $Q(x) \stackrel{\text{def}}{=} x(z).Q'(x,z)$

- The following action is enabled
  $P(x,y) \mid Q(x) \mid R(y) \xrightarrow{\tau} P'(x) \mid Q'(x,z)[y/z] \mid R(y)$

- Evolution of communication capabilities depicted graphically:



Example: encoding of data in the π calculus

- An encoding of True and False:

$$\mathit{False}(x) \stackrel{\text{def}}{=} !(\mathit{query}, \mathit{false}, \mathit{true})\ \overline{x}\ \mathit{query}.\overline{\mathit{query}}\ \mathit{false}.\overline{\mathit{query}}\ \mathit{true}.\overline{\mathit{false}}.0$$

$$\mathit{True}(x) \stackrel{\text{def}}{=} !(\mathit{query}, \mathit{false}, \mathit{true})\ \overline{x}\ \mathit{query}.\overline{\mathit{query}}\ \mathit{false}.\overline{\mathit{query}}\ \mathit{true}.\overline{\mathit{true}}.0$$

Example: encoding of data in the π calculus

- An encoding of True and False:

$$False(x) \stackrel{\text{def}}{=} !(query, false, true)\ \overline{x}\ query.\overline{query}\ false.\overline{query}\ true.\overline{false}.0$$
$$True(x) \stackrel{\text{def}}{=} !(query, false, true)\ \overline{x}\ query.\overline{query}\ false.\overline{query}\ true.\overline{true}.0$$

- Lets define a process $P(x)$ that behaves as P_1 if its argument x represents true and P_2 if it represents false:

$$P(x) \stackrel{\text{def}}{=} x(query).query(false).query(true).\,(true.P_1 + false.P_2)$$

Facultad de Informática Universidad Politécnica de Madrid

Example: encoding of data in the π calculus

- An encoding of True and False:

$$False(x) \stackrel{\text{def}}{=} !(query, false, true)\ \overline{x}\ query.\overline{query}\ false.\overline{query}\ true.\overline{false}.0$$
$$True(x) \stackrel{\text{def}}{=} !(query, false, true)\ \overline{x}\ query.\overline{query}\ false.\overline{query}\ true.\overline{true}.0$$

- Lets define a process $P(x)$ that behaves as P_1 if its argument x represents true and P_2 if it represents false:

$$P(x) \stackrel{\text{def}}{=} x(query).query(false).query(true).\,(true.P_1 + false.P_2)$$

- If we execute $P(x) \mid True(x)$ we will eventually end up in the new state $P_1 \mid 0 \mid True(x) = P_1 \mid True(x)$

Facultad de Informática
Universidad Politécnica de Madrid

Example: encoding of data in the π calculus

- An encoding of True and False:

$$False(x) \stackrel{\text{def}}{=} !(query, false, true)\ \overline{x}\ query.\overline{query}\ false.\overline{query}\ true.\overline{false}.0$$
$$True(x) \stackrel{\text{def}}{=} !(query, false, true)\ \overline{x}\ query.\overline{query}\ false.\overline{query}\ true.\overline{true}.0$$

- Lets define a process $P(x)$ that behaves as P_1 if its argument x represents true and P_2 if it represents false:

$$P(x) \stackrel{\text{def}}{=} x(query).query(false).query(true).\,(true.P_1 + false.P_2)$$

- If we execute $P(x) \mid True(x)$ we will eventually end up in the new state $P_1 \mid 0 \mid True(x) = P_1 \mid True(x)$
- The syntax is ugly; it is better in the polyadic π-calculus:

$$False(x) \stackrel{\text{def}}{=} !(false, true)\ \overline{x}\ \langle false, true\rangle\ .\overline{false}.0$$
$$P(x) \stackrel{\text{def}}{=} x(false, true).\,(true.P_1 + false.P_2)$$

π-calculus transition rules

$$\text{act}\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{sum}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\text{par}\ \frac{P \xrightarrow{\alpha} P' \qquad bn(\alpha) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{repl}\ \frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

$$\text{equiv}\ \frac{Q \xrightarrow{\alpha} Q' \qquad P \equiv Q \qquad P' \equiv Q'}{P \xrightarrow{\alpha} P'}$$

The rules uses the congruence $\equiv$ which is defined:

- $P \mid Q == Q \mid P$
- $P + Q \equiv Q + P$
- $[x = x]P \equiv P$
- if $A(x) \stackrel{\text{def}}{=} P'$ then $A(y) \equiv P'[y/x]$
- $P \equiv Q$ if P and Q are α-equivalent, i.e., only bound variables are different, e.g., $(x)\overline{y}\,x.0 \equiv (z)\overline{y}\,z.0$

Facultad de Informática
Universidad Politécnica de Madrid

Transition rules, part II

l-com $$\frac{P \xrightarrow{\bar{x}\, y} P' \qquad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'[y/z]}$$

close $$\frac{P \xrightarrow{\bar{x}(y)} P' \qquad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\alpha} (y)(P' \mid Q')}$$

res $$\frac{P \xrightarrow{\alpha} P' \qquad y \notin n(\alpha)}{(y)P \xrightarrow{\alpha} (y)P'}$$

open $$\frac{P \xrightarrow{\bar{x}\, y} P' \qquad y \neq x}{(y)P \xrightarrow{x(y)} P'}$$

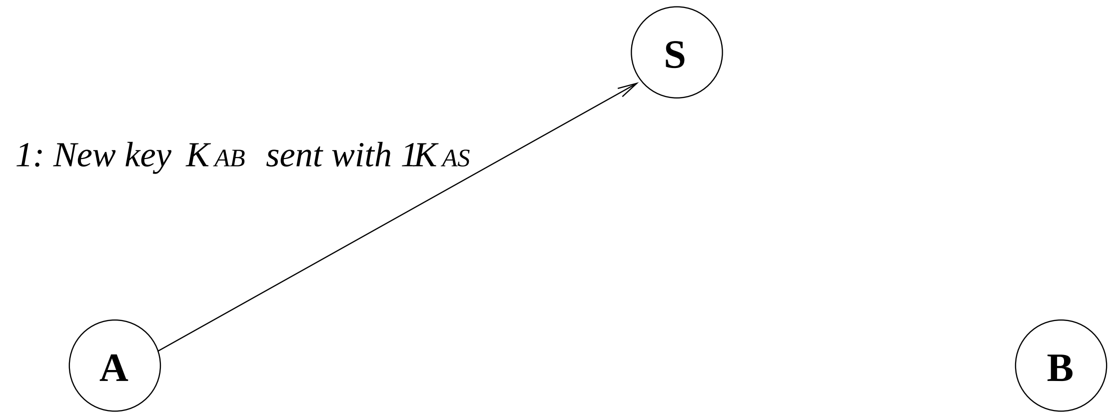
π-calculus: variants and implementations

- Asynchronous π-calculus
  Only 0 allowed after an output prefix
  $\overline{x}\,u.0$ is ok, $\overline{x}\,u.x(z).0$ is not!
- Higher-order π-calculus:
  communicating of processes as well as names
- spi-calculus
  A variant of the π-calculus for reasoning about security
- Ambient calculus: a process algebra for reasoning about distribution
- Pict: a programming language based on the asynchronous π-calculus
- WS-CDL: a web choreography language inspired by the π-calculus

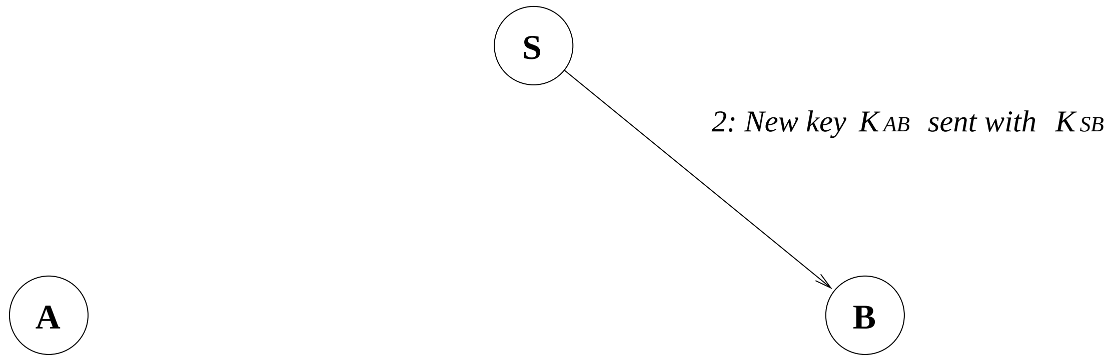
π-calculus variant: the spi-calculus

- spi-calculus: a variant of the π-calculus for reasoning about security
- Specially suited for reasoning about shared key cryptography
- Extends the normal π-calculus with a few new primitives:
  - $\{M\}N$ represents the ciphertext obtained by encrypting M under the key N
  - *case* L *of* $\{x\}N$ *in* P attempts to decrypt the term L with the key N. If L is a ciphertext of the form {M}N, then the process behaves as P[M/x]. Otherwise, the process is stuck.
  - ...
- The normal operational semantics of the π-calculus is extended, i.e., we get a lot of reasoning power for free

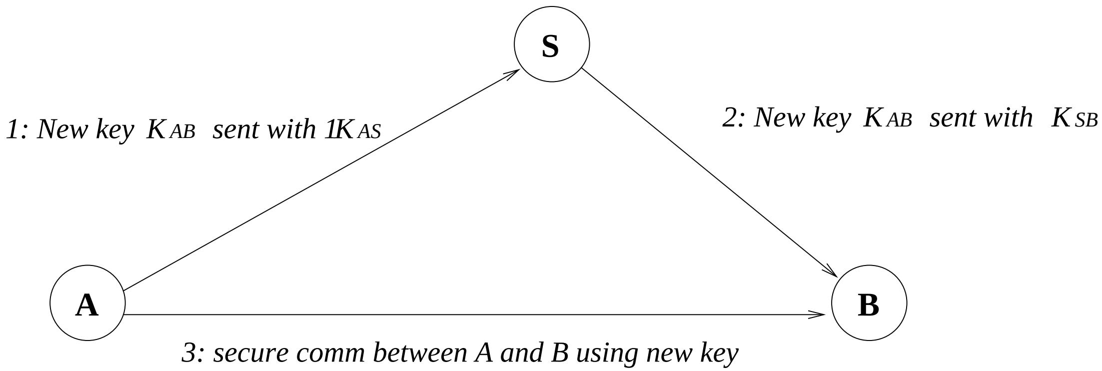
Spi Example: Wide Mouthed Frog protocol



Spi Example: Wide Mouthed Frog protocol



Spi Example: Wide Mouthed Frog protocol

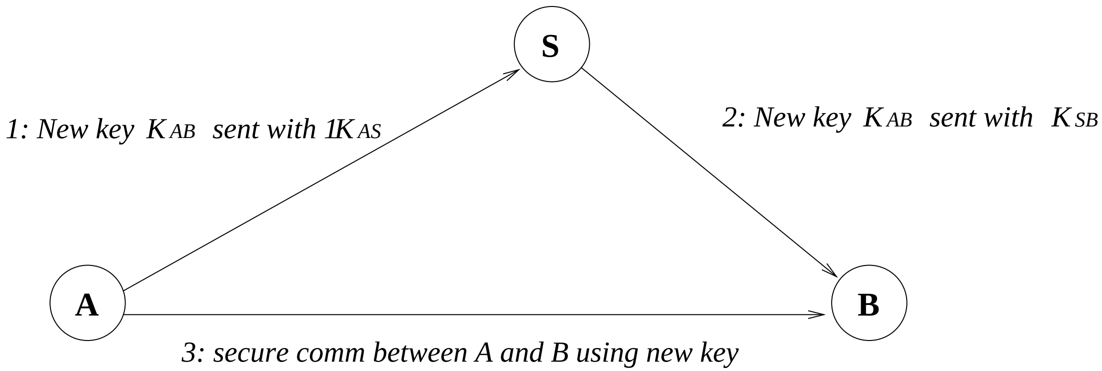


Message 1: A $\longrightarrow$ S: $\{K_{AB}\}K_{AS}$ on c_{AS}

Message 2: S $\longrightarrow$ B: $\{K_{AB}\}K_{SB}$ on c_{SB}

Message 3: A $\longrightarrow$ B: $\{M\}K_{AB}$ on c_{AB}

Spi Example: Wide Mouthed Frog protocol



Message 1: $A \rightarrow S$: $\{K_{AB}\}K_{AS}$ on c_{AS}

Message 2: $S \rightarrow B$: $\{K_{AB}\}K_{SB}$ on c_{SB}

Message 3: $A \rightarrow B$: $\{M\}K_{AB}$ on c_{AB}

In the spi-calculus:

$$A(M) \equiv \nu(K_{AB})(\overline{c_{AS}}\langle\{K_{AB}\}K_{AS}\rangle.\overline{c_{AB}}\langle\{M\}K_{AB}\rangle)$$

$$S \equiv c_{AS}(x).case\ x\ of\ \{y\}K_{AS}\ in\ \overline{c_{SB}}\langle\{y\}K_{SB}\rangle$$

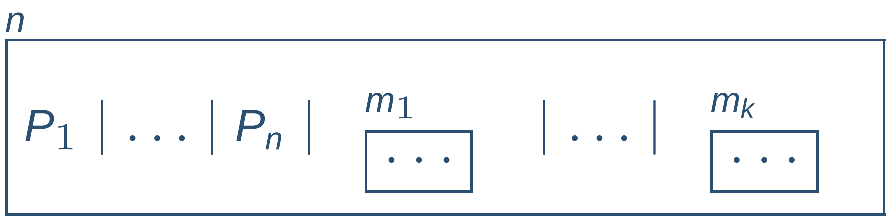
$$B \equiv c_{SB}(x).case\ x\ of\ \{y\}K_{SB}\ in\ c_{AB}(z).case\ z\ of\ \{w\}y\ in\ F(w)$$

Part I: The Ambient Calculus

- Due to Cardelli and Walker
- For modelling mobile computation – mobile code that moves between locations (ambients)
- Used to reason about administrative domains – when does a program have the right to migrate from one computing location to another computing location and start computing there?
- An *ambient* is a bounded place where computation takes place
- An ambient can be nested in other ambients
- Each ambient has a name used to control access, and a set of local agents (processes) that control the actions of the ambient
- A *name* is something that can be created, communicated and from which capabilities can be extracted

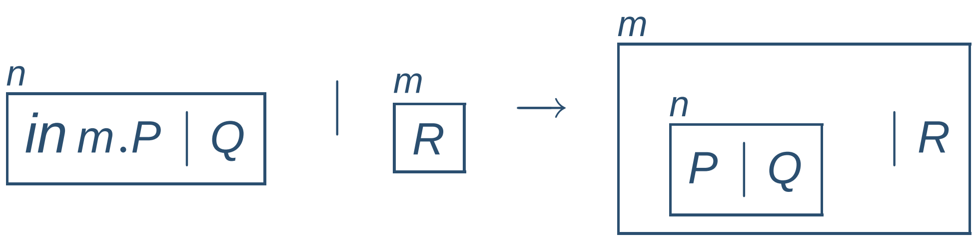
Ambient Calculus: operators

- As in the π-calculus: 0 (the process which can do nothing), $P \mid Q$ – parallel composition, replication $!P$ and creation of a new name n in $(n)P$.
- An ambient is written $n[P]$
  where n is the name and P is the process running inside the ambient
- If $P \longrightarrow P'$ then $n[P] \longrightarrow n[P']$
- The general shape of an ambient is $n[P_1 \mid \ldots \mid P_n \mid m_1[\ldots] \mid \ldots \mid m_k[\ldots]]$ where P_i is a non-ambient process and $m_i[\ldots]$ is a subambient of n
- In graphical notation:



Ambient Calculus: operators

- An action prefix is written $M.P$, where M enters, exits or opens an ambient
- Entry capability: *in* $m.P$ instructs the ambient surrounding *in* $m.P$ to enter a sibling named m



- Exit capability: *out* $m.P$ instructs the ambient surrounding *out* $m.P$ to exit its parent named m

Open capability and communication

- Open capability: $\mathit{open}\, m.P$ which provides a way of dissolving the boundary of an ambient m located at the same level as $\mathit{open}\, m.P$

$$\mathit{open}\, m.P \mid \overset{m}{\boxed{Q}} \longrightarrow P \mid Q$$

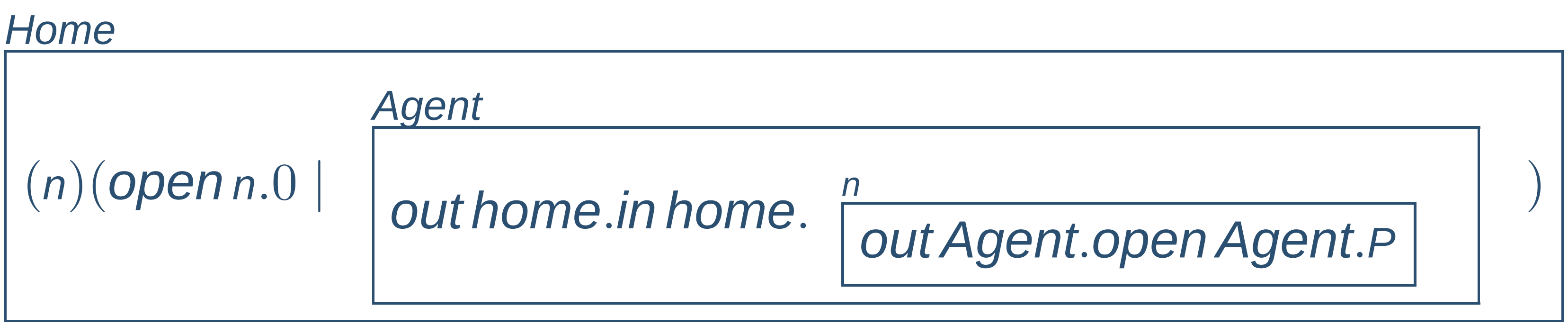
- The full calculus adds two items:
  - Capabilities can be paths $M.M'$
  - A communication rule between processes in the same ambient: $(x)P \mid \langle M \rangle \longrightarrow P[M/x]$ where $(x)P$ is an input prefix and $\langle M \rangle$ an output

Ambient example

Motivating example: an agent P leaves its home ambient and later comes back (with authentification)

$$
\mathit{Home}\ \Big[\ (n)\Big(\ \mathit{open}\ n.0\ \mid\ \mathit{Agent}[\mathit{out}\ \mathit{home}.\mathit{in}\ \mathit{home}.n[\mathit{out}\ \mathit{Agent}.\mathit{open}\ \mathit{Agent}.P]]\ \Big)\ \Big]
$$

or graphically:



Evaluation of the Example

$$Home[(n)\ (open\ n.0 \mid Agent[out\ home.in\ home.n[out\ Agent.open\ Agent.P]])]$$

Evaluation of the Example

$Home[(n)\ (open\ n.0 \mid Agent[out\ home.in\ home.n[out\ Agent.open\ Agent.P]])]$

$\equiv \quad (n)Home[open\ n.0 \mid$
$Agent[out\ home.in\ home.n[out\ Agent.open\ Agent.P]]]$

Evaluation of the Example

$Home[(n)\ (open\ n.0 \mid Agent[out\ home.in\ home.n[out\ Agent.open\ Agent.P]])]$

$\equiv \quad (n)Home[open\ n.0 \mid Agent[out\ home.in\ home.n[out\ Agent.open\ Agent.P]]]$

$\downarrow out\ home$

$(n)\ (Home[open\ n.0] \mid Agent[in\ home.n[out\ Agent.open\ Agent.P]])$

Facultad de Informática
Universidad Politécnica de Madrid

Evaluation of the Example

$$Home[(n)\ (open\ n.0 \mid Agent[out\ home.in\ home.n[out\ Agent.open\ Agent.P]])]$$

$$\equiv \quad (n)Home[open\ n.0 \mid Agent[out\ home.in\ home.n[out\ Agent.open\ Agent.P]]]$$

$$\downarrow out\ home$$

$$(n)\ (Home[open\ n.0] \mid Agent[in\ home.n[out\ Agent.open\ Agent.P]])$$

$$\downarrow in\ home$$

$$(n)\ (Home[open\ n.0 \mid Agent[n[out\ Agent.open\ Agent.P]]])$$

Facultad de Informática Universidad Politécnica de Madrid

Evaluation of the Example

$$Home[(n)\,(open\; n.0 \mid Agent[out\; home.in\; home.n[out\; Agent.open\; Agent.P]])]$$

$$\equiv \quad (n)Home[open\; n.0 \mid$$
$$Agent[out\; home.in\; home.n[out\; Agent.open\; Agent.P]]]$$

$$\downarrow out\; home$$
$$(n)\,(Home[open\; n.0] \mid Agent[in\; home.n[out\; Agent.open\; Agent.P]])$$

$$\downarrow in\; home$$
$$(n)\,(Home[open\; n.0 \mid Agent[n[out\; Agent.open\; Agent.P]]])$$

$$\downarrow out\; Agent$$
$$(n)\,(Home[open\; n.0 \mid n[open\; Agent.P] \mid Agent[]])$$

Evaluation of the Example

$Home[(n)\ (open\ n.0 \mid Agent[out\ home.in\ home.n[out\ Agent.open\ Agent.P]])]$

$\equiv \quad (n)Home[open\ n.0 \mid Agent[out\ home.in\ home.n[out\ Agent.open\ Agent.P]]]$

$\downarrow$ *out home*

$(n)\ (Home[open\ n.0] \mid Agent[in\ home.n[out\ Agent.open\ Agent.P]])$

$\downarrow$ *in home*

$(n)\ (Home[open\ n.0 \mid Agent[n[out\ Agent.open\ Agent.P]]])$

$\downarrow$ *out Agent*

$(n)\ (Home[open\ n.0 \mid n[open\ Agent.P] \mid Agent[]])$

$\downarrow$ *open n*

$(n)\ (Home[0 \mid open\ Agent.P \mid Agent[]])$

Facultad de Informática
Universidad Politécnica de Madrid

Evaluation of the Example

$$Home[(n)\,(open\,n.0 \mid Agent[out\,home.in\,home.n[out\,Agent.open\,Agent.P]])]$$

$$\equiv \quad (n)Home[open\,n.0 \mid Agent[out\,home.in\,home.n[out\,Agent.open\,Agent.P]]]$$

$$\downarrow out\ home$$

$$(n)\,(Home[open\,n.0] \mid Agent[in\,home.n[out\,Agent.open\,Agent.P]])$$

$$\downarrow in\ home$$

$$(n)\,(Home[open\,n.0 \mid Agent[n[out\,Agent.open\,Agent.P]]])$$

$$\downarrow out\ Agent$$

$$(n)\,(Home[open\,n.0 \mid n[open\,Agent.P] \mid Agent[]])$$

$$\downarrow open\ n$$

$$(n)\,(Home[0 \mid open\,Agent.P \mid Agent[]])$$

$$\downarrow open\ Agent$$

$$(n)\,(Home[0 \mid P \mid 0]) \quad \equiv \quad Home[P]$$

Facultad de Informática
Universidad Politécnica de Madrid

Tools for CCS and the π-calculus

There exists a set of tools for CCS and the π-calculus that permits to:

- simulate specifications,
- to check whether two specifications are equivalent,
- to check a specification against a property in temporal logic, and so on. . .

Tools:

- For CCS: Concurrency workbench, Concurrency Workbench of the New Century (CWB-NC), . . .
- For π-calculus: Mobility workbench, . . .