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
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

Design Phase

Programming

Testing
Formal Methods in a Project Design Cycle

Project Phases

Design Phase

Programming

Testing

Artifact

Design Specification

Program

Test specification
Formal Methods in a Project Design Cycle

<table>
<thead>
<tr>
<th>Project Phases</th>
<th>Artifact</th>
<th>Formal or informal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Phase</td>
<td><em>Design Specification</em></td>
<td>Normally informal,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>but formalizable</td>
</tr>
<tr>
<td>Programming</td>
<td><em>Program</em></td>
<td>Already formal!</td>
</tr>
<tr>
<td>Testing</td>
<td><em>Test specification</em></td>
<td>A set of tests,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>normally a mix of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>formal and informal</td>
</tr>
</tbody>
</table>

- p. 4
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
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
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
- 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):
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:

\{\text{Pre}\} \ Command \ \{\text{Post}\}
Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:
  \[ \{ \text{Pre} \} \text{ Command } \{ \text{Post} \} \]

- Example proof rules:

\[
\frac{\{ c \land l \} \quad \text{body} \quad \{ l \} \quad}{\{ l \} \quad \text{while } c \text{ do } \text{body} \quad \{ \neg c \land l \}}
\]

\[
\frac{\phi' \supset \phi \quad \psi \supset \psi' \quad \{ \phi \} \ C \ \{ \psi \} \quad}{\{ \phi' \} \ C \ \{ \psi' \}}
\]
Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:
  \{ Pre \} \textit{Command} \{ Post \}

- Example proof rules:

- An example proof:
Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:
  \[
  \{ \text{Pre} \} \quad \text{Command} \quad \{ \text{Post} \}
  \]

- Example proof rules:

  \[
  \begin{align*}
  \{ C \land I \} & \quad \text{body} \quad \{ I \} \\
  \{ I \} & \quad \text{while} \ C \ \text{do} \ \text{body} \quad \{ \neg C \land I \} \\
  \phi' & \supset \phi \\
  \psi & \supset \psi' \\
  \{ \phi \} \ C \ \{ \psi \} \\
  \{ \phi' \} \ C \ \{ \psi' \}
  \end{align*}
  \]

- An example proof:

  \[
  \begin{align*}
  \{ i = N \land j = 0 \} \quad &\text{while} \ i > 0 \ \text{do} \ i := i - 1; \ j := j + 1 \quad \{ j = N \}
  \end{align*}
  \]
Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:
  \{Pre\} Command \{Post\}

- Example proof rules:

\[
\begin{align*}
\{ C \land I \} & \quad \text{body} \quad \{ I \} \\
\{ I \} & \quad \text{while} \ C \ \text{do} \ \text{body} \quad \{ \neg C \land I \} \\
& \quad \phi' \supset \phi \quad \psi \supset \psi' \quad \{ \phi \} \ C \ \{ \psi \} \\
& \quad \{ \phi' \} \ C \ \{ \psi' \}
\end{align*}
\]

- An example proof:

\[
\begin{align*}
\{ j = N - i \} & \quad \text{while} \ i > 0 \ \text{do} \ i := i - 1; j := j + 1 \quad \{ i = 0 \land j = N - i \} \\
& \quad i = N \land j = 0 \supset j = N - i \quad i = 0 \land j = N - i \supset j = N \\
& \quad \{ i = N \land j = 0 \} \quad \text{while} \ i > 0 \ \text{do} \ i := i - 1; j := j + 1 \quad \{ j = N \}
\end{align*}
\]
Formalisations – History

- **Hoare logic**: putting pre-and-post conditions in syntactic form:
  \[ \{Pre\} \text{ Command } \{Post\} \]

- **Example proof rules**:
  \[
  \begin{align*}
  \{C \land I\} \quad \text{body} \quad \{I\} \\
  \{I\} \quad \text{while } C \text{ do } \text{body} \quad \{\neg C \land I\}
  \end{align*}
  \]

  \[
  \begin{align*}
  \phi' \supset \phi \quad \psi \supset \psi' \quad \{\phi\} \quad C \quad \{\psi\}
  \end{align*}
  \]

- **An example proof**:
  \[
  \begin{align*}
  \{i > 0 \land j = N - i\} \quad &i := i - 1; \quad j := j + 1 \quad \{j = N - i\} \\
  \{j = N - i\} \quad &\text{while } i > 0 \text{ do } i := i - 1; \quad j := j + 1 \quad \{i = 0 \land j = N - i\} \\
  \quad &i = N \land j = 0 \supset j = N - i \quad i = 0 \land j = N - i \supset j = N \\
  \{i = N \land j = 0\} \quad &\text{while } i > 0 \text{ do } i := i - 1; \quad j := j + 1 \quad \{j = N\}
  \end{align*}
  \]
Formalisations – History

- Hoare logic: putting pre-and-post conditions in syntactic form:
  \{Pre\} Command \{Post\}

- Example proof rules:

  \[
  \begin{align*}
  \{C \land l\} & \quad \text{body} \quad \{l\} \\
  \{l\} & \quad \text{while } C \text{ do } \text{body} \quad \{\neg C \land l\} \\
  \phi' & \supset \phi \\
  \psi & \supset \psi' \\
  \{\phi'\} & \quad C \quad \{\psi'\}
  \end{align*}
  \]

- An example proof:

  \[
  \begin{align*}
  \models (N - i) + 1 &= N - (i - 1) \\
  \{i > 0 \land j = N - i\} & \quad i := i - 1; \quad j := j + 1 \quad \{j = N - i\} \\
  \{j = N - i\} & \quad \text{while } i > 0 \text{ do } i := i - 1; \quad j := j + 1 \quad \{i = 0 \land j = N - i\} \\
  i = N \land j = 0 & \supset j = N - i \\
  i = 0 \land j = N - i & \supset j = N \\
  \{i = N \land j = 0\} & \quad \text{while } i > 0 \text{ do } i := i - 1; \quad j := j + 1 \quad \{j = N\}
  \end{align*}
  \]
Formalisations – History

- Dijkstra: weakest pre condition: \( \text{wp}(C) \)

- 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 system 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
Consider the following simple shared variable program:

\[
\text{if } x > 0 \text{ then } x := x - 1 \mid \mid \text{if } x < 3 \text{ then } x := x + 1
\]

where \( S_1 \mid \mid S_2 \) runs the atomic statements \( S_1 \) and \( S_2 \) in parallel
Runs of concurrent programs: examples

Consider the following simple shared variable program:

\[
\text{if } x > 0 \text{ then } x := x - 1 \quad || \quad \text{if } x < 3 \text{ then } x := x + 1
\]

where \( S_1 || S_2 \) runs the atomic statements \( S_1 \) and \( S_2 \) in parallel.

Its runs starting from the state \( x = 0 \) is the infinite set:

\[
\begin{align*}
\langle x = 0 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle & \cdot \ldots \\
\langle x = 0 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle & \cdot \ldots \\
\langle x = 0 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 3 \rangle \cdot \langle x = 2 \rangle & \cdot \ldots \\
\ldots
\end{align*}
\]
We can also depict the runs

\[
\begin{align*}
\langle x = 0 \rangle & \cdot \langle x = 1 \rangle & \cdot \langle x = 0 \rangle & \cdot \ldots \\
\langle x = 0 \rangle & \cdot \langle x = 1 \rangle & \cdot \langle x = 2 \rangle & \cdot \langle x = 1 \rangle & \cdot \ldots \\
\langle x = 0 \rangle & \cdot \langle x = 1 \rangle & \cdot \langle x = 2 \rangle & \cdot \langle x = 3 \rangle & \cdot \langle x = 2 \rangle & \cdot \ldots \\
\ldots
\end{align*}
\]

as a state graph:
Atomicity

Consider the parallel program

\[
\text{if } x > 0 \text{ then } x := x - 1 \quad \| \quad \text{if } x > 0 \text{ then } x := x - 1
\]

with the starting state \( \langle x = 3 \rangle \)
Atomicity

Consider the parallel program

\[
\text{if } x > 0 \text{ then } x := x - 1 \ | \ | \text{ if } x > 0 \text{ then } x := x - 1
\]

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

\[
\text{if } x > 0 \text{ then } x := x - 1 \ | \ | \ \text{if } x > 0 \text{ then } x := x - 1
\]

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

\[ \text{if } x > 0 \text{ then } x := x - 1 \quad \text{||} \quad \text{if } x > 0 \text{ then } x := x - 1 \]

with the starting state \( \langle x = 3 \rangle \)

- If statements are \textbf{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 \textbf{not atomic} the program has the two runs

\[
\begin{align*}
\langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \\
\langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \cdot \langle x = -1 \rangle
\end{align*}
\]

- \textbf{Conclusion:} programming concurrent programs is \textit{hard}
Atomicity

Consider the parallel program

\[
\text{if } x > 0 \text{ then } x := x - 1 \quad \mid \quad \text{if } x > 0 \text{ then } x := x - 1
\]

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
  
  \[
  \begin{cases}
  \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \\
  \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \cdot \langle x = -1 \rangle
  \end{cases}
  \]

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

- The *Erlang programming language* we shall hear about soon has **good** concurrency primitives
Temporal logic – operators

Classical linear temporal operators (defined over runs):

- **Always** $\phi$
  $\phi$ holds in all future states of the run

- **Eventually** $\phi$
  $\phi$ holds in some future state of the run

- **Next** $\phi$
  $\phi$ holds in the next state

- $\phi_1$ **Until** $\phi_2$
  $\phi_1$ holds in all states until $\phi_2$ holds

- And the normal ones: negation $\neg \phi$, conjunction $\phi_1 \land \phi_2$, implication $\phi_1 \supset \phi_2$, . . .

- And propositional operators (over variables): $x < y$, even($z$), . . .
Temporal logic – meaning

- A program $p$ satisfies a formula $\phi$ 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

\[
\text{if } x > 0 \text{ then } x := x - 1 \quad || \quad \text{if } x < 3 \text{ then } x := x + 1
\]

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

\[\begin{array}{cccc}
x = 0 & \rightarrow & x = 1 & \rightarrow & x = 2 & \rightarrow & x = 3 \\
\end{array}\]
Temporal logic – examples

Consider the atomic parallel program

\[
\text{if } x > 0 \text{ then } x := x - 1 \; \| \; \text{if } x < 3 \text{ then } x := x + 1
\]

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

Does \textit{Always} \( x \geq 0 \) hold?
Temporal logic – examples

Consider the atomic parallel program

\[
\text{if } x > 0 \text{ then } x := x - 1 \mid \mid \text{ if } x < 3 \text{ then } x := x + 1
\]

with the starting state \( x = 3 \) and the state graph

- Does \textit{Always} \( x \geq 0 \) hold?
- \textbf{Yes}; if \( x = 0 \) then the guard prevents further decrease
Consider the atomic parallel program

\[
\text{if } x > 0 \text{ then } x := x - 1 \ || \ \text{if } x < 3 \text{ then } x := x + 1
\]

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

- Does \textbf{Always } \( x \geq 0 \) hold?
- \textbf{Yes}; if \( x = 0 \) then the guard prevents further decrease
- Does \textbf{Always } \( (x = 3 \supset \text{Eventually } x = 0) \) hold?
Temporal logic – examples

Consider the atomic parallel program

\[
\text{if } x > 0 \text{ then } x := x - 1 \quad \| \quad \text{if } x < 3 \text{ then } x := x + 1
\]

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 ($\neg \phi$)
General temporal logic patterns

- Eventually $\phi \equiv \neg \text{Always } (\neg \phi)$

- A safety property expresses that something bad $-\phi$ never happens: Always $\neg \phi$
General temporal logic patterns

- Eventually $\phi \equiv \neg \text{Always } (\neg \phi)$

- A safety property expresses that something bad $- \phi$ never happens: Always $\neg \phi$

- A liveness property expresses that something good $\phi$ eventually happens: Eventually $\phi$
General temporal logic patterns

- *Eventually* \( \phi \equiv \neg \text{Always } (\neg \phi) \)

- A *safety property* expresses that something bad – \( \phi \) – never happens: \( \text{Always } \neg \phi \)

- A *liveness property* expresses that something good – \( \phi \) – eventually happens: \( \text{Eventually } \phi \)

- Often one uses *fairness assumptions* to rule out bad program behaviours
  \( \phi \) eventually holds under the assumption that \( \psi \) doesn’t always hold:
  \[
  (\neg \text{Always } \psi) \supset (\text{Eventually } \phi)
  \]
Temporal logic

Many variants exist:
- Continuous time, time interval models
- Additional operators (e.g., talking about past time)
- Branching time logics: $\mathcal{A} \phi$ – $\phi$ 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...
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, $\pi$ 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

\[
\begin{align*}
\text{Result} & ::= \text{ok} \mid \text{nospace} \\
\text{Queue} & \\
\quad \text{queue} : \text{seq} \mathbb{N} \\
\quad \#queue & < 10 \\
\text{Init} & \\
\quad \text{Queue} & \\
\quad \quad \text{queue} & = \langle \rangle
\end{align*}
\]
Z schema example, dynamic part

\[\begin{align*}
\text{InsertOk} & \\
\Delta \text{Queue} & \\
\text{insert}^? : \mathbb{N} & \\
\text{result}! : \text{Result} & \\
\#\text{queue} < 10 & \\
\text{queue}' = \text{queue} \smallsetminus \langle \text{insert}^? \rangle & \\
\text{result}! = \text{ok} & \\
\end{align*}\]

\[\begin{align*}
\text{InsertWithError} & \\
\Xi \text{Queue} & \\
\text{insert}^? : \mathbb{N} & \\
\text{result}! : \text{Result} & \\
\#\text{queue} \geq 10 & \\
\text{result}! = \text{nospace} & \\
\end{align*}\]

\[\text{Insert} = \text{InsertOk} \lor \text{InsertWithError}\]
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:

\[
\begin{align*}
\text{BadOp} & \quad \text{BadOp} \\
\text{Δ Queue} & \quad \Delta \text{Queue} \\
\text{insert} & : \mathbb{N} \\
\text{result!} & : \text{Result} \\
\text{queue} &= \text{queue} \uplus \langle \text{insert}\? \rangle \\
\text{result!} &= \text{ok}
\end{align*}
\]

Violates the state invariant
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/](http://maude.cs.uiuc.edu/)
- **Origin at Stanford, USA**
  (Meseguer 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 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?

  \[ \text{red head}(\text{enc}(2,\text{enc}(3,\text{empty}))) \]
Usage examples

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

\[
\text{red head(\text{enc}(2,\text{enc}(3,\text{empty}))))}.
\]

■ 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:

  \[
  \begin{array}{ccc}
  \epsilon & 1 \\
  2 & 1 \\
  2 \cdot 2 & 1 \cdot 1 & 2 \cdot 1 \\
  2 \cdot 2 \cdot 1 & 2 \cdot 1 \cdot 1 \\
  2 \cdot 2 \cdot 2 \cdot 1 & 2 \cdot 2 \cdot 1 \cdot 1 & 2 \cdot 1 \cdot 1 \cdot 1 \\
  \end{array}
  \]

- 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
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.
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, $\pi$-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: $\pi$-calculus
  - With extensions to distribution: mobile ambients
  - With abstract data types: LOTOS, $\mu$CRL
  - 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: \((a\text{ is a port})\)
  - output action \(\overline{a}(v)\)
  - input action \(a(x)\)
  - or internal action \(\tau\)

  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 \(\alpha\) is an action (input or output action, or internal \(\tau\))
  - **Behaviour**: after performing \(a\) it behaves as \(P\)

- **Choice**: \(P + Q\) can behave as \(P\) or as \(Q\)

- **Parallel behaviour**: the agent \(P | Q\) 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:

\[
\text{coin.} \left( \overline{\text{coffee}.0} + \overline{\text{tea}.0} \right)
\]
A simple example: a coffee/tea machine

- A simple (one-use) coffee machine:

\[
\text{coin. \left( \text{coffee.0 + tea.0} \right)}
\]

- A user that always wants tea:

\[
\text{coin.tea.0}
\]
A simple example: a coffee/tea machine

- A simple (one-use) coffee machine:
  
  \[ \text{coin}. \left( \text{coffee}.0 + \text{tea}.0 \right) \]

- A user that always wants tea:
  
  \[ \text{coin}. \text{tea}.0 \]

- The combination of a user and the coffee machine:
  
  \[ \text{coin}. \left( \text{coffee}.0 + \text{tea}.0 \right) \mid \text{coin}. \text{tea}.0 \]
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: $P \overset{\text{def}}{=} a.(P \mid b.0)$

- And simple test on boolean conditions: $\text{if } b \text{ then } P \text{ else } Q$
A simple example: a coffee/tea machine continued

- A simple coffee machine:

\[ CM1 \overset{\text{def}}{=} \text{coin.} \left( \text{coffee}.CM1 + \text{tea}.CM1 \right) \]

- A user that always wants tea:

\[ User \overset{\text{def}}{=} \text{coin.tea.}0 \]

- The combination of a user and the coffee machine:

\( (User \mid CM1) \setminus \{\text{coffee, tea, coin}\} \)
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** \( \alpha \cdot P \xrightarrow{\alpha} P \)

- **choice** \( P \xrightarrow{\alpha} P' \), \( Q \xrightarrow{\alpha} Q' \), \( P + Q \xrightarrow{\alpha} P' \), \( Q + P \xrightarrow{\alpha} P' \)

- **interleaving** \( P \xrightarrow{\alpha} P' \), \( Q \xrightarrow{\alpha} Q' \), \( P \parallel Q \xrightarrow{\alpha} P' \parallel Q' \), \( Q \parallel P \xrightarrow{\alpha} Q \parallel P' \)

- **synchronisation** \( P \xrightarrow{a(x)} P' \), \( Q \xrightarrow{\overline{a}(v)} Q' \), \( P \parallel Q \xrightarrow{\tau} P'[v/x] \parallel Q' \)

- **synchronisation** \( Q \xrightarrow{a(x)} Q' \), \( P \xrightarrow{\overline{a}(v)} P' \), \( P \parallel Q \xrightarrow{\tau} P'[v/x] \parallel Q' \)
CCS, Operational behaviour part II

- **restriction**
  \[ P \xrightarrow{\alpha} P' \quad a \notin fn(\alpha) \]
  \[ P \setminus \{a\} \xrightarrow{\alpha} P' \setminus \{a\} \]
  where \( fn(\alpha) \equiv \begin{cases} 
  \{a\} & \text{if } \alpha = \overline{a}(v) \\
  \{a\} & \text{if } \alpha = a(x) \\
  \{\} & \text{if } \alpha = \tau 
  \end{cases} \)

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

- **recursion**
  \[ P[v/x] \xrightarrow{\alpha} P' \]
  \[ N(x) \xrightarrow{\alpha} P' \]
  \[ N(v) \xrightarrow{\alpha} P' \]

- **eval**
  \[ P \xrightarrow{\alpha} P' \quad b \text{ true} \]
  \[ if \ b \ then \ P \ else \ Q \xrightarrow{\alpha} P' \]
  \[ P \xrightarrow{\alpha} P' \quad b \text{ false} \]
  \[ if \ b \ then \ Q \ else \ P \xrightarrow{\alpha} P' \]
CCS examples

- Coffee machine 1 always work:
  \[ CM1 \overset{\text{def}}{=} \text{coin.} \left( \overline{\text{coffee}.CM1} + \overline{\text{tea}.CM1} \right) \]
CCS examples

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

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

- Coffee machine 1 always work:
  \[ CM1 \overset{\text{def}}{=} \text{coin.} \left( \text{coffee}.CM1 + \text{tea}.CM1 \right) \]

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

- A typical user that always wants tea:
  \[ User \overset{\text{def}}{=} \text{coin}.\text{tea}.0 \]
CCS examples

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

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

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

- The combination of a user and machine 1 (let $$S \equiv \{\text{coffee, tea, coin}\}$$):
  $$(User \mid CM1) \setminus S \xrightarrow{\tau} \xrightarrow{\tau} (0 \mid CM1) \setminus S$$
CCS examples

- Coffee machine 1 always work:
  \[ CM1 \overset{\text{def}}{=} \text{coin. } (\text{coffee}.CM1 + \text{tea}.CM1) \]

- Coffee machine 2 sometimes (unspecified when) only allows the coffee choice:
  \[ CM2 \overset{\text{def}}{=} \text{coin. } (\text{coffee}.CM2 + \text{tea}.CM2 + \tau.\text{coffee}.CM2) \]

- A typical user that always wants tea:
  \[ User \overset{\text{def}}{=} \text{coin.} \text{tea}.0 \]

- The combination of a user and machine 1 (let \( S \equiv \{ \text{coffee, tea, coin} \} \)):
  \[ (User \mid CM1) \setminus S \xrightarrow{\tau} \xrightarrow{\tau} (0 \mid CM1) \setminus S \]

- The combination of a user and machine 2 may deadlock after two machine steps:
  \[ (User \mid CM2) \setminus S \xrightarrow{\tau} \xrightarrow{\tau} (\text{tea}.0 \mid \text{coffee}.CM2) \setminus S \]
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 $\phi$, and show that $P : \phi$ ($P$ satisfies $\phi$)

- Write the specification as another CCS agent $S$, and show that $P = S$, with regards to some notion of 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 practice, this is often too weak, e.g., $CM_1$ and (a slight variant of) $CM_2$ 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 $P + Q = Q + P$, $P | 0 = P$ 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 processes $P$ 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 $\alpha$ 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 $\alpha$ 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
### $\pi$ calculus operators

- Most operators come from CCS: $0$ – the inactive process, choice $P + Q$, parallelism $P \mid Q$

- Communication primitives are different:
  - **output**: $\overline{x} y.P$: 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:
  \[ P(x, y) \overset{\text{def}}{=} \overline{x} \, y. \, P'(x) \quad \text{and} \quad Q(x) \overset{\text{def}}{=} x(z). \, Q'(x, z) \]

- **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:**

![Diagram](attachment:image.png)
Example: encoding of data in the $\pi$ calculus

- An encoding of True and False:

\[
\begin{align*}
\text{False}(x) & \overset{\text{def}}{=} ! (query, \text{false}, \text{true}) \times query.\overline{query} \text{false.} \overline{query} \text{true.false.} 0 \\
\text{True}(x) & \overset{\text{def}}{=} ! (query, \text{false}, \text{true}) \times query.\overline{query} \text{false.} \overline{query} \text{true.true.} 0
\end{align*}
\]
Example: encoding of data in the $\pi$ calculus

- An encoding of True and False:

\[
\begin{align*}
\text{False}(x) & \overset{\text{def}}{=} !(\text{query}, \text{false}, \text{true}) \times \text{query}.\text{query}.\text{false}.\text{query}.\text{true}.\text{false}.0 \\
\text{True}(x) & \overset{\text{def}}{=} !(\text{query}, \text{false}, \text{true}) \times \text{query}.\text{query}.\text{false}.\text{query}.\text{true}.\text{true}.0
\end{align*}
\]

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

\[
P(x) \overset{\text{def}}{=} x(\text{query}).\text{query}(\text{false}).\text{query}(\text{true}).(\text{true}.P_1 + \text{false}.P_2)
\]
Example: encoding of data in the $\pi$ calculus

- An encoding of True and False:

$$\begin{align*}
\text{False}(x) & \overset{\text{def}}{=} !(\text{query}, \text{false}, \text{true}) \times \text{query.}\overline{\text{query}} \text{false.}\overline{\text{query}} \text{true.}\overline{\text{false}}.0 \\
\text{True}(x) & \overset{\text{def}}{=} !(\text{query}, \text{false}, \text{true}) \times \text{query.}\overline{\text{query}} \text{false.}\overline{\text{query}} \text{true.}\overline{\text{true}}.0
\end{align*}$$

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

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

- If we execute $P(x) \mid \text{True}(x)$ we will eventually end up in the new state $P_1 \mid 0 \mid \text{True}(x) = P_1 \mid \text{True}(x)$
Example: encoding of data in the $\pi$ calculus

- An encoding of True and False:

\[
\begin{align*}
\text{False}(x) & \overset{\text{def}}{=} !(\text{query}, \text{false}, \text{true}) \times \text{query.} \text{false.} \text{true.} \text{false.} .0 \\
\text{True}(x) & \overset{\text{def}}{=} !(\text{query}, \text{false}, \text{true}) \times \text{query.} \text{false.} \text{true.} \text{true.} .0
\end{align*}
\]

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

\[
P(x) \overset{\text{def}}{=} x(\text{query}). \text{query}(\text{false}). \text{query}(\text{true}). (\text{true}. P_1 + \text{false}. P_2)
\]

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

- The syntax is ugly; it is better in the polyadic $\pi$-calculus:

\[
\begin{align*}
\text{False}(x) & \overset{\text{def}}{=} !(\text{false, true}) \times \langle \text{false, true} \rangle. \text{false.} .0 \\
P(x) & \overset{\text{def}}{=} x(\text{false, true}). (\text{true}. P_1 + \text{false}. P_2)
\end{align*}
\]
\( \pi \)-calculus transition rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>act</td>
<td>( \alpha.P \xrightarrow{\alpha} P )</td>
</tr>
<tr>
<td>sum</td>
<td>( P \xrightarrow{\alpha} P' )</td>
</tr>
<tr>
<td>par</td>
<td>( P \xrightarrow{\alpha} P' )</td>
</tr>
<tr>
<td>bn</td>
<td>( bn(\alpha) \cap fn(Q) = \emptyset )</td>
</tr>
<tr>
<td>repl</td>
<td>( P \mid P \xrightarrow{\alpha} P' )</td>
</tr>
</tbody>
</table>

The rules uses the congruence \( \equiv \) which is defined:

- \( P \mid Q \equiv Q \mid P \)
- \( P + Q \equiv Q + P \)
- \([x = x]P \equiv P\)
- if \( A(x) \overset{\text{def}}{=} P' \) then \( A(y) \equiv P'[y/x] \)
- \( P \equiv Q \) if \( P \) and \( Q \) are \( \alpha \)-equivalent, i.e., only bound variables are different, e.g., \((x)y x.0 \equiv (z)y z.0\)
Transition rules, part II

I-com

\[
\begin{align*}
P & \xrightarrow{\overline{\lambda} y} P' \\
Q & \xrightarrow{\lambda(z)} Q' \\
\hline
P | Q & \xrightarrow{\tau} P' | Q'[y/z]
\end{align*}
\]

close

\[
\begin{align*}
P & \xrightarrow{\overline{\lambda}(y)} P' \\
Q & \xrightarrow{\lambda(y)} Q' \\
\hline
P | Q & \xrightarrow{\alpha} (y)(P' | Q')
\end{align*}
\]

res

\[
\begin{align*}
P & \xrightarrow{\alpha} P' \\
y & \not\in n(\alpha) \\
\hline
(y)P & \xrightarrow{\alpha} (y)P'
\end{align*}
\]

open

\[
\begin{align*}
P & \xrightarrow{\overline{\lambda} y} P' \\
y & \not\equiv x \\
\hline
(y)P & \xrightarrow{\lambda(y)} P'
\end{align*}
\]
\[\pi\text{-calculus: variants and implementations}\]

- **Asynchronous \(\pi\)-calculus**
  Only 0 allowed after an output prefix
  \(\bar{x} \ u \ . \ 0\) is ok, \(\bar{x} \ u \ . \ x(z) \ . \ 0\) is not!

- **Higher-order \(\pi\)-calculus:**
  communicating of processes as well as names

- **spi-calculus**
  A variant of the \(\pi\)-calculus for reasoning about security

- **Ambient calculus**: a process algebra for reasoning about distribution

- **Pict**: a programming language based on the asynchronous \(\pi\)-calculus

- **WS-CDL**: a web choreography language inspired by the \(\pi\)-calculus
\(\pi\)-calculus variant: the spi-calculus

- spi-calculus: a variant of the \(\pi\)-calculus for reasoning about security
- Specially suited for reasoning about shared key cryptography
- Extends the normal \(\pi\)-calculus with a few new primitives:
  - \(\{M\}N\) represents the ciphertext obtained by encrypting \(M\) under the key \(N\)
  - \textit{case} \(L\) \textit{of} \(\{x\}N\) \textit{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.
  - \(\ldots\)
- The normal operational semantics of the \(\pi\)-calculus is extended, i.e., we get a lot of reasoning power for free
Spi Example: Wide Mouthed Frog protocol

1: New key $K_{AB}$ sent with $IK_{AS}$
Spi Example: Wide Mouthed Frog protocol

2: New key $K_{AB}$ sent with $K_{SB}$
Spi Example: Wide Mouthed Frog protocol

1: New key $K_{AB}$ sent with $IK_{AS}$

2: New key $K_{AB}$ sent with $K_{SB}$

3: secure comm between $A$ and $B$ using new key

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 $c_{AB}$
Spi Example: Wide Mouthed Frog protocol

Message 1:  A → S: \( \{K_{AB}\}K_{AS} \) on \( c_{AS} \)
Message 2:  S → B: \( \{K_{AB}\}K_{SB} \) on \( c_{SB} \)
Message 3:  A → B: \( \{M\}K_{AB} \) on \( c_{AB} \)

In the spi-calculus:

\[
A(M) \equiv \nu(K_{AB}) (c_{AS} (\{K_{AB}\}K_{AS}) \cdot c_{AB} (\{M\}K_{AB})) \\
S \equiv c_{AS}(x) . \text{case } x \text{ of } \{y\}K_{AS} \text{ in } c_{SB}(\{y\}K_{SB}) \\
B \equiv c_{SB}(x) . \text{case } x \text{ of } \{y\}K_{SB} \text{ in } c_{AB}(z) . \text{case } z \text{ of } \{w\}y \text{ in } F(w)
\]
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 $\pi$-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 $P \rightarrow P'$ then $n[P] \rightarrow n[P']$.

- 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:

\[
\begin{array}{c}
\text{n} \\
\hline
P_1 \mid \ldots \mid P_n \mid \begin{array}{c}
\ldots \\
\ldots
\end{array} \\
\hline
m_1 \mid \ldots \mid m_k
\end{array}
\]
Ambient Calculus: operators

- An action prefix is written $M.P$, where $M$ enters, exits or opens an ambient

- **Entry capability:** $\textit{in} m. P$ instructs the ambient surrounding $\textit{in} m. P$ to enter a sibling named $m$

- **Exit capability:** $\textit{out} m. P$ instructs the ambient surrounding $\textit{out} m. P$ to exit its parent named $m$
Open capability and communication

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

  $$\text{open } m.P \mid \begin{array}{c} m \\ Q \end{array} \rightarrow P \mid Q$$

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

\[
\text{Home} \quad \left[ \begin{array}{c}
(n) \\
\left( \begin{array}{c}
\text{open } n.0 \\
\text{Agent[out home.in home.n[out Agent.open Agent.P]]}
\end{array} \right)
\end{array} \right]
\]

or graphically:
Evaluation of the Example

\[
\text{Home}[n \ (\text{open} \ n \ 0 \ | \ \text{Agent}[\text{out home.in home.n[\text{out Agent.open Agent.P}]]})]
\]
Evaluation of the Example

\[
\text{Home}[(n) \ (\text{open } n.0 \ | \ \text{Agent}[\text{out home.in home.n[\text{out Agent.open Agent.P}]]})]\]
\equiv \ (n)\text{Home}[\text{open } n.0 \ | \ \text{Agent}[\text{out home.in home.n[\text{out Agent.open Agent.P}]]}]]
Evaluation of the Example

\[ \text{Home}[ (n) \ (\text{open } n.0 \ | \ \text{Agent}[\text{out home.in home.n}[\text{out Agent.open Agent.P}]]]) ] \]

\[ \equiv \ (n) \ \text{Home}[\text{open } n.0 \ | \ \text{Agent}[\text{out home.in home.n}[\text{out Agent.open Agent.P}]]] \]

\[ \downarrow \text{out home} \]

\[ (n) \ (\text{Home}[\text{open } n.0] \ | \ \text{Agent}[\text{in home.n}[\text{out Agent.open Agent.P}]])) \]
Evaluation of the Example

\[ \text{Home}[(n) \ (\text{open} \ n \cdot 0 \ | \ \text{Agent}[\text{out home}.\text{in home}.\ n[\text{out Agent}.\text{open Agent}.P]]))] \]

\[ \equiv \ (n) \text{Home}[\text{open} \ n \cdot 0 \ | \ \text{Agent}[\text{out home}.\text{in home}.\ n[\text{out Agent}.\text{open Agent}.P]]] \]

\[ \downarrow \text{out home} \]

\[ (n) \ (\text{Home}[\text{open} \ n \cdot 0] \ | \ \text{Agent}[\text{in home}.\ n[\text{out Agent}.\text{open Agent}.P]])) \]

\[ \downarrow \text{in home} \]

\[ (n) \ (\text{Home}[\text{open} \ n \cdot 0] \ | \ \text{Agent}[n[\text{out Agent}.\text{open Agent}.P]])) \]
Evaluation of the Example

\[
\text{Home}[(n) \ (\text{open} \ n \ 0 \ | \ \text{Agent}[\text{out home}. \text{in home}. \ n[\text{out Agent}. \text{open Agent}. \ P]])]
\equiv \ (n) \text{Home}[\text{open} \ n \ 0 \ | \ \text{Agent}[\text{out home}. \text{in home}. \ n[\text{out Agent}. \text{open Agent}. \ P]])
\]

\[
\downarrow \text{out home}
\]

\[
(n) \ (\text{Home}[\text{open} \ n \ 0] \ | \ \text{Agent}[\text{in home}. \ n[\text{out Agent}. \text{open Agent}. \ P]])
\]

\[
\downarrow \text{in home}
\]

\[
(n) \ (\text{Home}[\text{open} \ n \ 0] \ | \ \text{Agent}[n[\text{out Agent}. \text{open Agent}. \ P]])
\]

\[
\downarrow \text{out Agent}
\]

\[
(n) \ (\text{Home}[\text{open} \ n \ 0] \ | \ n[\text{open Agent}. \ P] \ | \ \text{Agent}[])
\]
Evaluation of the Example

\[
\begin{align*}
& \text{Home}[(n) \ (\text{open } n.0 \ | \ \text{Agent}[\text{out home.in home.n[\text{out Agent.open Agent.P}]]})] \\
\equiv & \ (n) \text{Home[open } n.0 \ | \\
& \text{Agent[out home.in home.n[\text{out Agent.open Agent.P}]]}] \\
\downarrow & \text{out home} \\
& (n) \ (\text{Home[open } n.0 \ | \ \text{Agent[in home.n[\text{out Agent.open Agent.P}]]})] \\
\downarrow & \text{in home} \\
& (n) \ (\text{Home[open } n.0 \ | \ \text{Agent[n[\text{out Agent.open Agent.P}]]}]]) \\
\downarrow & \text{out Agent} \\
& (n) \ (\text{Home[open } n.0 \ | \ n[\text{open Agent.P} \ | \ \text{Agent[]}]]) \\
\downarrow & \text{open n} \\
& (n) \ (\text{Home[0 | open Agent.P | Agent[]}])
\end{align*}
\]
Evaluation of the Example

\[ \text{Home}[(n) \ (\text{open } n.0 \ | \ \text{Agent}[\text{out home.in home.n[\text{out Agent.open Agent.P}]]})] \]

\[ \equiv \ (n)\text{Home}[\text{open } n.0 \ | \ \text{Agent}[\text{out home.in home.n[\text{out Agent.open Agent.P}]]}] \]

\[ \downarrow \text{out home} \]

\[ (n) \ (\text{Home}[\text{open } n.0 \ | \ \text{Agent}[\text{in home.n[\text{out Agent.open Agent.P}]]})] \]

\[ \downarrow \text{in home} \]

\[ (n) \ (\text{Home}[\text{open } n.0 \ | \ \text{Agent}[n[\text{out Agent.open Agent.P}]]]) \]

\[ \downarrow \text{out Agent} \]

\[ (n) \ (\text{Home}[\text{open } n.0 \ | \ n[\text{open Agent.P} \ | \ \text{Agent[]}]) \]

\[ \downarrow \text{open } n \]

\[ (n) \ (\text{Home}[0 \ | \ \text{open Agent.P} \ | \ \text{Agent[]}]) \]

\[ \downarrow \text{open Agent} \]

\[ (n) \ (\text{Home}[0 \ | \ P \ | \ 0]) \equiv \text{Home}[P] \]
Tools for CCS and the $\pi$-calculus

There exists a set of tools for CCS and the $\pi$-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 $\pi$-calculus: Mobility workbench, . . .