Non-Sequential Theory of Distributed Systems

Lecturers

* Benedikt Bollig (responsible)
* Paul Gastin

Dates

The lectures will take place on Wednesday, 13:15–15:45, in Bat. Sophie Germain, room 1004.
The first lecture is on Wednesday, 12th September 2018.

Outline of the course

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>12.09.2018</td>
<td>motivating examples, automata models of concurrent processes with data structures</td>
</tr>
<tr>
<td>2nd</td>
<td></td>
<td>operational semantics; graph semantics; MSO logic</td>
</tr>
<tr>
<td>3rd</td>
<td></td>
<td>expressive power of MSO logic; underapproximate verification</td>
</tr>
<tr>
<td>4th</td>
<td></td>
<td>special tree-width, decomposition game</td>
</tr>
<tr>
<td>5th</td>
<td></td>
<td>decision procedures via tree automata</td>
</tr>
<tr>
<td>6th</td>
<td></td>
<td>propositional dynamic logic (PDL)</td>
</tr>
<tr>
<td>7th</td>
<td></td>
<td>translation of PDL formulas into CPDS</td>
</tr>
<tr>
<td>8th</td>
<td></td>
<td>asynchronous automata</td>
</tr>
<tr>
<td>9th</td>
<td></td>
<td>revision and exercises</td>
</tr>
<tr>
<td>Exam</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lecture Notes

non-sequential.pdf (version as of 31st October 2017, covering lectures 1–7)

Prerequisites

Basic knowledge in Automata, Logics, Complexity.

While not mandatory, it is useful to know the basics of verification. See for instance the level 1 course 1–22 Basics of verification which can also be taken during M2.
Landscape & Objectives

\[ \neg F \]

specification

\[ \rightarrow \]

distributed system
Landscape & Objectives

specification

System model

\( \varphi \rightarrow A \)
Landscape & Objectives

- **Landscape:**
  - **Specification** $L(\varphi)$
  - **System model** $L(A)$

- **Objectives:**
  - **Behavior**
  - **set of possible executions**
  - **set of admissible executions**

- **Model Checking:**
  - $L(\varphi) \supseteq L(A)$?

- **Graphical Representation:**
  - Directed graph showing relationships between $L(\varphi)$, $\varphi$, $L(A)$, and $A$. 
  - Arrows indicate flow, with $L(\varphi)$ pointing to $\varphi$, and $L(A)$ pointing to $A$. 
  - Labels include model checking and set inclusion conditions.
Landscape & Objectives

Specification

System model

$L(\varphi) \rightarrow \text{Behavior} \rightarrow L(A)$

realizability

$\exists A: L(\varphi) = L(A)$?
### Various settings

<table>
<thead>
<tr>
<th>Communication</th>
<th>Architecture</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>single process</td>
<td>static &amp; known</td>
<td>finite state</td>
</tr>
<tr>
<td>shared memory</td>
<td>static &amp; unknown (parameterized)</td>
<td>recursive</td>
</tr>
<tr>
<td>message passing</td>
<td>dynamic</td>
<td>timed</td>
</tr>
</tbody>
</table>
Finite automata

**Behavior**
- Words

**System model**
- Finite automata

**Specification**
- Linear-time temporal logic (LTL)
- Monadic second-order logic (MSO)

**Communication**
- Single process
- Shared memory
- Message passing

**Architecture**
- Static & known
- Static & unknown (parameterized)
- Dynamic

**Process**
- Finite state
- Recursive
- Timed
Asynchronous automata

**System model**
- Asynchronous automata
  - Zielonka '87

**Specification**
- Linear-time temporal logic (LTL)
- Monadic second-order logic (MSO)

**Behavior**
- Mazurkiewicz traces
  - Mazurkiewicz '70s

**Architecture**
- single process
- shared memory
- message passing
- static & known
- static & unknown (parameterized)
- dynamic

**Communication**
- finite state
- recursive
- timed
Communicating automata

**Behavior**
- Message sequence charts

**System model**
- Communicating automata
  - [Brand-Zafiropulo ‘83]

**Specification**
- Linear-time temporal logic (LTL)
- Monadic second-order logic (MSO)

**Communication**
- single process
- shared memory
- message passing

**Architecture**
- static & known
- static & unknown (parameterized)
- dynamic

**Process**
- finite state
- recursive
- timed
Multi-pushdown automata

communication
- single process
- shared memory
- message passing

architecture
- static & known
- static & unknown (parameterized)
- dynamic

process
- finite state
- recursive
- timed

Behavior
- (Multiply) Nested words

System_model
- Visibly multi-pushdown automata
  [Alur-Madhusudan '04]
  [La Torre-Madhusudan-Parlato '07]

Specification
- Linear-time temporal logic (LTL)
- Monadic second-order logic (MSO)
Data automata

- Communication:
  - Single process
  - Shared memory
  - Message passing

- Architecture:
  - Static & known
  - Static & unknown (parameterized)
  - Dynamic

- Process:
  - Finite state
  - Recursive
  - Timed

- Behavior:
  - Data words (over infinite alphabet)

- System model:
  - Data/Class-memory automata
    - [Bojanczyk et al. '06]
    - [Björklund-Schwentick'07]

- Specification:
  - Linear-time temporal logic (LTL)
  - Monadic second-order logic (MSO)

Many different models …
Graph automata

**Behavior**
- Graphs/Partial orders

**System model**
- Graph automata

**Specification**
- Linear-time temporal logic (LTL)
- Monadic second-order logic (MSO)

... in a uniform framework.
Landscape & Objectives

Behavior

{ executions }

Specification

\( \varphi \)

System model

\( \mathcal{A} \)

realizability

\[ \exists A: L(\varphi) = L(\mathcal{A}) \ ? \]

model checking

\[ L(\varphi) \supseteq L(\mathcal{A}) \ ? \]
Motivating examples
synchronous communication
(rendez-vous, handshake)
1.1.2 Algorithms

A first, rather obvious solution would consist in using timestamps. Strictly increasing timestamps are sent by $D$ along with a status message, forwarded by $P_1$ and $P_2$, and then compared by $L$ with its latest knowledge.

It is easy to see that the protocol is correct in the sense that $L$ does not update its display based on outdated information. The following scenario shows that the error from the previous solution is indeed avoided.
Protocol is correct.  
But unbounded counters are needed.  

How about finite-state solution (i.e., with a bounded number of messages)?  
Difficult to design.  
Impossible in our example (no « feedback »).
A solution exists for:

… but it is still very difficult to obtain. (When can a time stamp be reused?)

Is there an automatic way of synthesizing a distributed program from a specification?
Here are some example words to illustrate:

In our example, the specification is contained in an MSO formula.

\[ L_{\text{spec}} \subseteq \Sigma^* \]

\[ \Sigma = \{ \langle \text{on} \rightarrow \text{P1} \rangle, \langle \text{on} \rightarrow \text{P2} \rangle, \langle \text{off} \rightarrow \text{P1} \rangle, \langle \text{off} \rightarrow \text{P2} \rangle \} \]

\[ \cup \{ \langle \text{on} \rightarrow \text{L} \rangle, \langle \text{on} \rightarrow \text{off} \rightarrow \text{L} \rangle, \langle \text{off} \rightarrow \text{L} \rangle, \langle \text{L} \rightarrow \text{L} \rangle \} \]

\[ \cup \{ \text{on}, \text{off}, \text{noop} \} \]

\[ \langle \text{on} \rightarrow \text{P1} \rangle \langle \text{on} \rightarrow \text{L} \rangle \langle \text{off} \rightarrow \text{P1} \rangle \text{ on} \langle \text{on} \rightarrow \text{P2} \rangle \langle \text{on} \rightarrow \text{P2} \rangle \text{ on} \langle \text{off} \rightarrow \text{P1} \rangle \text{noop} \]
\[ L_{\text{spec}} \subseteq \Sigma^* \]

\[ \Sigma = \{ \langle D \xRightarrow{\text{on}} P1, \langle D \xRightarrow{\text{on}} P2, \langle D \xRightarrow{\text{off}} P1, \langle D \xRightarrow{\text{off}} P2 \rangle \} \]

\[ \cup \{ \langle P1 \xRightarrow{\text{on}} L, \langle P2 \xRightarrow{\text{on}} L, \langle P1 \xRightarrow{\text{off}} L, \langle P2 \xRightarrow{\text{off}} L \rangle \} \]

\[ \cup \{ \text{on, off, noop} \} \]

1. The projection to P1 is in:

\[ (\langle D \xRightarrow{\text{on}} P1 \langle P1 \xRightarrow{\text{on}} L, + \langle D \xRightarrow{\text{off}} P1 \langle P1 \xRightarrow{\text{off}} L \rangle)^* \]
\[ L_{spec} \subseteq \Sigma^* \]

\[
\Sigma = \{ \langle D \rightleftharpoons P1 \rangle, \langle D \rightleftharpoons P2 \rangle, \langle D \rightleftharpoons P1 \rangle, \langle D \rightleftharpoons P2 \rangle \} \\
\cup \{ \langle P1 \rightleftharpoons L \rangle, \langle P2 \rightleftharpoons L \rangle, \langle P1 \rightleftharpoons L \rangle, \langle P2 \rightleftharpoons L \rangle \} \\
\cup \{ on, off, noop \}
\]

1. The projection to P1 is in:

\[ (\langle D \rightleftharpoons P1 \rangle \langle P1 \rightleftharpoons L \rangle + \langle D \rightleftharpoons P1 \rangle \langle P1 \rightleftharpoons L \rangle)^* . \]

2. Similarly for P2.
1. The projection to P1 is in:
\[
(\langle D \leftrightarrow P1 \rangle \langle P1 \leftrightarrow L \rangle + \langle D \leftrightarrow P1 \rangle \langle P1 \leftrightarrow L \rangle)^*.
\]

2. Similarly for P2.

3. The projection to L is in:
\[
\left( (\langle P1 \leftrightarrow L \rangle + \langle P2 \leftrightarrow L \rangle + \langle P1 \leftrightarrow L \rangle + \langle P2 \leftrightarrow L \rangle)(on + off + noop) \right)^*.
\]
\[ L_{spec} \subseteq \Sigma^* \]

\[ \Sigma = \{ \langle D \rightarrow P1 \rangle, \langle D \rightarrow P2 \rangle, \langle D \rightarrow P1 \rangle, \langle D \rightarrow P2 \rangle \} \]

\[ \cup \{ \langle P1 \rightarrow L \rangle, \langle P2 \rightarrow L \rangle, \langle P1 \rightarrow L \rangle, \langle P2 \rightarrow L \rangle \} \]

\[ \cup \{ on, off, noop \} \]

1. The projection to \( P1 \) is in:

\[ (\langle D \rightarrow P1 \rangle \langle P1 \rightarrow L \rangle + \langle D \rightarrow P1 \rangle \langle P1 \rightarrow L \rangle)^* \]

2. Similarly for \( P2 \).

3. The projection to \( L \) is in:

\[ ((\langle P1 \rightarrow L \rangle + \langle P2 \rightarrow L \rangle + \langle P1 \rightarrow L \rangle + \langle P2 \rightarrow L \rangle)(on + off + noop))^* \]

4. The display is updated iff the last (previous) status message emitted by \( D \) that has already been followed by a forward was not yet followed by a corresponding update by \( L \).
Here are some example words to illustrate Exercise 1.2.

Obviously, with this, let an MSO formula.

The projection of $P_1$. Here, two actions are said to be on $P_1$.

$D$ is a regular language. Moreover, it is closed under on $P_2$.

Let $\{\langle P_1 \rightarrow L \rangle, \langle P_2 \rightarrow L \rangle, \langle P_1 \leftarrow L \rangle, \langle P_2 \leftarrow L \rangle\}$

The projection of $D$ on $P_1$.

Let $\{on, off, noop\}$

\[
\begin{align*}
\Sigma = & \{\langle D \rightarrow P_1 \rangle, \langle D \rightarrow P_2 \rangle, \langle D \leftarrow P_1 \rangle, \langle D \leftarrow P_2 \rangle\} \\
\cup & \{\langle P_1 \rightarrow L \rangle, \langle P_2 \rightarrow L \rangle, \langle P_1 \leftarrow L \rangle, \langle P_2 \leftarrow L \rangle\} \\
\cup & \{on, off, noop\}
\end{align*}
\]

\[
\langle D \rightarrow P_1 \rangle \langle P_1 \rightarrow L \rangle \langle D \rightarrow P_1 \rangle \langle P_2 \rightarrow L \rangle \langle D \rightarrow P_2 \rangle \langle P_2 \rightarrow L \rangle \langle P_2 \leftarrow L \rangle \langle P_1 \rightarrow L \rangle \langle P_1 \leftarrow L \rangle \langle D \rightarrow P_1 \rangle \langle D \leftarrow P_1 \rangle \langle D \rightarrow P_2 \rangle \langle P_2 \rightarrow L \rangle \langle D \rightarrow P_2 \rangle \langle P_2 \leftarrow L \rangle \langle P_1 \leftarrow L \rangle \langle P_1 \rightarrow L \rangle \

1. The projection to $P_1$ is in:

\[
(\langle D \rightarrow P_1 \rangle \langle P_1 \rightarrow L \rangle + \langle D \rightarrow P_1 \rangle \langle P_1 \leftarrow L \rangle)^*.
\]

2. Similarly for $P_2$.

3. The projection to $L$ is in:

\[
\left((\langle P_1 \rightarrow L \rangle + \langle P_2 \rightarrow L \rangle + \langle P_1 \leftarrow L \rangle + \langle P_2 \leftarrow L \rangle)(on + off + noop)\right)^*.
\]

4. The display is updated iff the last (previous) status message emitted by $D$ that has already been followed by a forward was not yet followed by a corresponding update by $L$. 
\[ L_{\text{spec}} \subseteq \Sigma^* \]

\[
\Sigma = \{ \langle D \rightleftharpoons P1 \rangle, \langle D \rightleftharpoons P2 \rangle, \langle D \rightleftharpoons P1 \rangle, \langle D \rightleftharpoons P2 \rangle \} \\
\quad \cup \{ \langle P1 \rightleftharpoons L \rangle, \langle P2 \rightleftharpoons L \rangle, \langle P1 \rightleftharpoons L \rangle, \langle P2 \rightleftharpoons L \rangle \} \\
\quad \cup \{ \text{on, off, noop} \}
\]

1. The projection to P1 is in:
\[
(\langle D \rightleftharpoons P1 \rangle \langle P1 \rightleftharpoons L \rangle + \langle D \rightleftharpoons P1 \rangle \langle P1 \rightleftharpoons L \rangle)^*. 
\]

2. Similarly for P2.

3. The projection to L is in:
\[
\left( (\langle P1 \rightleftharpoons L \rangle + \langle P2 \rightleftharpoons L \rangle + \langle P1 \rightleftharpoons L \rangle + \langle P2 \rightleftharpoons L \rangle)(\text{on} + \text{off} + \text{noop}) \right)^*. 
\]

4. The display is updated iff the last (previous) status message emitted by D that has already been followed by a forward was not yet followed by a corresponding update by L.
Exercise: What are the words contained in $L_{spec} \subseteq \Sigma^*$?

- $\langle \text{on} \rangle \langle \text{off} \rangle \langle \text{off} \rangle \langle \text{on} \rangle \langle \text{noop} \rangle \langle \text{noop} \rangle$ ✓
Exercise: What are the words contained in $L_{spec} \subseteq \Sigma^*$?

- $\langle D \leftrightarrow P1 \rangle \langle D \check{\leftrightarrow} P2 \rangle \langle P2 \leftrightarrow L \rangle \check{\leftrightarrow} \langle P1 \leftrightarrow L \rangle \text{noop \checkmark}$
- $\langle D \leftrightarrow P1 \rangle \langle P1 \leftrightarrow L \rangle \langle D \leftrightarrow P1 \rangle \text{on \times}$
Exercise: What are the words contained in $L_{\text{spec}} \subseteq \Sigma^*$?

- $\langle D \overset{\text{on}}{\Rightarrow} P1 \overset{\text{off}}{\Rightarrow} D \overset{\text{off}}{\Rightarrow} P2 \overset{\text{on}}{\Rightarrow} P2 \overset{\text{off}}{\Rightarrow} L \overset{\text{off}}{\Rightarrow} P1 \overset{\text{on}}{\Rightarrow} L \overset{\text{noop}}{\Rightarrow} \rangle$ ✓
- $\langle D \overset{\text{on}}{\Rightarrow} P1 \overset{\text{on}}{\Rightarrow} P1 \overset{\text{on}}{\Rightarrow} L \overset{\text{on}}{\Rightarrow} D \overset{\text{on}}{\Rightarrow} P1 \overset{\text{on}}{\Rightarrow} \rangle$ ✗
- $\langle D \overset{\text{on}}{\Rightarrow} P1 \overset{\text{on}}{\Rightarrow} P1 \overset{\text{on}}{\Rightarrow} L \overset{\text{on}}{\Rightarrow} D \overset{\text{off}}{\Rightarrow} P2 \overset{\text{on}}{\Rightarrow} P2 \overset{\text{off}}{\Rightarrow} L \overset{\text{off}}{\Rightarrow} \rangle$ ✓
Exercise: What are the words contained in $L_{spec} \subseteq \Sigma^*$?

- $\langle D \leftrightarrow P1 \rangle \langle D \leftrightarrow P2 \rangle \langle P2 \leftrightarrow L \rangle^{off} \langle P1 \leftrightarrow L \rangle^{noop}$ ✓
- $\langle D \leftrightarrow P1 \rangle \langle P1 \leftrightarrow L \rangle \langle D \leftrightarrow P1 \rangle^{on}$ ✗
- $\langle D \leftrightarrow P1 \rangle \langle P1 \leftrightarrow L \rangle \langle D \leftrightarrow P2 \rangle^{off} \langle P2 \leftrightarrow L \rangle^{off}$ ✓
- $\langle D \leftrightarrow P1 \rangle \langle P1 \leftrightarrow L \rangle^{on} \langle D \leftrightarrow P2 \rangle^{off} \langle P2 \leftrightarrow L \rangle^{off}$ ✓
\[ L_{\text{spec}} \subseteq \Sigma^* \]

\[
\Sigma = \{ \langle D \xrightarrow{on} P1 \rangle, \langle D \xrightarrow{on} P2 \rangle, \langle D \xrightarrow{off} P1 \rangle, \langle D \xrightarrow{off} P2 \rangle \} \\
\cup \{ \langle P1 \xrightarrow{on} L \rangle, \langle P2 \xrightarrow{on} L \rangle, \langle P1 \xrightarrow{off} L \rangle, \langle P2 \xrightarrow{off} L \rangle \} \\
\cup \{ \text{on, off, noop} \}
\]

\[
\langle D \xrightarrow{on} P1 \rangle \langle P1 \xrightarrow{on} L \rangle \langle D \xrightarrow{off} P1 \rangle \langle D \xrightarrow{on} P2 \rangle \langle P2 \xrightarrow{on} L \rangle \langle P1 \xrightarrow{off} L \rangle \text{noop}
\]

**Observation:**
- \( L_{\text{spec}} \) is regular.
- \( L_{\text{spec}} \) is closed under permutation rewriting of independent events.

**Theorem [Zielonka 1987]:**

Let \( L \) be a regular set of words that is closed under permutation rewriting of independent events. There is a deterministic finite-state distributed protocol that realizes \( L \).

\[
L = (\langle D \xrightarrow{on} P1 \rangle \langle P2 \xrightarrow{on} L \rangle)^* \text{ is not realizable.}
\]
4. The display is updated if the last (previous) status message emitted by D that has already been followed by a forward was not yet followed by a corresponding update by L.

\[
\langle D \leftarrow P1 \rangle \langle P1 \rightarrow L \rangle \langle D \leftarrow P2 \rangle \langle D \leftarrow P1 \rangle \langle P1 \rightarrow L \rangle \langle P2 \leftarrow L \rangle \text{noop} \in L_{\text{spec}}
\]

no causal dependency

last event in partial order

consider partial orders
An advantage of MSO logic that is directly interpreted over partial orders is the exercise of giving an MSO formula for (R4').

Suppose this partial order is denoted by \( L \).

Let \( L \) be an MSO-definable set of partial orders. There is a finite-state distributed protocol that realizes \( L \).

Theorem [Thomas 1990]:

Let \( L \) be an MSO-definable set of partial orders. There is a finite-state distributed protocol that realizes \( L \).

“When \( L \) performs on (off, respectively), then the last (wrt. \( \leq \)) status message sent by D should also be on (off, respectively). Moreover, a display operation should be noop iff there has already been an acknowledgement between the latest status message and that operation.”
Reasoning about recursive processes
Consider the following protocol:

- From time to time, P1 sends requests to P2.
- When receiving a request, P2 calls a procedure, which itself may call sub-procedures.
- When returning from the (outermost) procedure, P2 sends an acknowledgment to P1.

\[
\langle \text{!req} \rangle \langle \text{?req} \rangle \langle \text{call} \rangle \langle \text{call} \rangle \langle \text{ret} \rangle \langle \text{call} \rangle \langle \text{ret} \rangle \langle \text{ret} \rangle \langle \text{!ack} \rangle \langle ?ack \rangle \quad \checkmark
\]

\[
\langle \text{!req} \rangle \langle \text{?req} \rangle \langle \text{call} \rangle \langle \text{call} \rangle \langle \text{ret} \rangle \langle \text{call} \rangle \langle \text{call} \rangle \langle \text{ret} \rangle \langle \text{ret} \rangle \langle \text{!ack} \rangle \langle ?ack \rangle \langle \text{ret} \rangle \quad \times
\]

Is this a regular language? No!

So, what is a good specification language?
We abstract the communication and for the storage and retrieval of data. These computers may access the network concurrently, which may communicate among themselves via shared memory. This models real-world systems like several computers connected with a receive event.

The previous discussion somehow motivates the naming of our lecture. However, the introductory example, we assumed synchronous communication, which roughly corresponds to a word, and hence an MSCN is a union of linear processes.

Moreover, we will consider stacks, FIFO channels, and bags (with the restriction that stacks are connected to data structures). The latter, when receiving a request, calls a procedure, performs some internal actions, and returns from the procedure, before it sends an acknowledgment. In other words, we add an acknowledgment immediately after returning from this subroutine. Thus, we shall have dependencies, but they provide even more information. For example, they may indicate via

\[ \text{msg} \]

dependencies.

The lecture is concerned with distributed systems with a fixed architecture: There connect a process with itself, with the purpose of modeling recursion. Thus, we have a network of multi-pushdown systems communicating via shared memory.

From this point of view, we can now express our property easily in a suitable MSO logic:

\[ \forall x \langle ?\text{req} \rangle(x) \Rightarrow \exists y_1, y_2, z \left( x \rightarrow y_1 \land \text{cr}(y_1, y_2) \land y_2 \rightarrow z \land \langle !\text{ack} \rangle(z) \right) \]

Consider a system of two processes, \( P_1 \) and \( P_2 \), connected by two unbounded channels. Cf. Cours 2.9.1. The behaviours of such systems may be modelled with a receive event. In other words, we add an acknowledgment immediately after returning from this subroutine. Thus, we shall have dependencies, but they provide even more information. For example, they may indicate via

\[ \text{msg} \]

dependencies.

Consider a system of two processes, \( P_1 \) and \( P_2 \), connected by two unbounded channels. Cf. Cours 2.9.1. The behaviours of such systems may be modelled with a receive event.
Let's be more formal …