Non-Sequential Theory of Distributed Systems

Lecture MPRI M2

Benedikt Bollig & Paul Gastin

November 30, 2015
## Contents

1 Introduction .................................................. 1  
  1.1 Synthesis and Control .................................... 1  
  1.2 Modeling behaviors as graphs ............................ 6  
  1.3 Underapproximate Verification ........................... 9  

2 Concurrent Processes with Data Structures ....... 12  
  2.1 The Model ................................................... 12  
  2.2 Operational Semantics ................................... 14  
  2.3 Graph Semantics .......................................... 16  

3 Logic ........................................................... 20  
  3.1 Monadic Second-Order Logic ............................... 20  
  3.2 Expressive Power of MSO Logic ............................ 22  
  3.3 Propositional Dynamic Logic ............................... 26  
  3.4 Expressive Power of ICPDL ................................. 29  
  3.5 Satisfiability and Model Checking ......................... 35  

4 Underapproximate Verification .................... 37  
  4.1 Principles of Underapproximate Verification ............ 37  
  4.2 Graph-Theoretic Approach ................................ 39  
  4.3 Graph (De)composition and Tree Interpretation ........ 40  
  4.4 Split-Width ............................................... 43
4.5 Main Results ................................................. 46
4.6 Special tree terms and tree interpretation ...................... 47
4.7 Split-width and Special tree-width ............................. 50
4.8 Tree automata and Decision procedures ....................... 52
4.9 ICPDL model checking .................................... 57

5 Distributed systems with shared resources ............... 66
  5.1 Architecture for shared resources ......................... 66
  5.2 Asynchronous automata .................................. 68
  5.3 Traces as graphs or partial orders ......................... 71
  5.4 Checking Closure of DFAs ................................ 77
  5.5 Star-Connected Regular Expressions ....................... 77
  5.6 Logic Specifications .................................... 78
Before we go into formal definitions, we give some examples of the kind of problems that are treated in the lecture.

1.1 Synthesis and Control

1.1.1 A simple protocol

In this lecture, we consider systems that consist of a fixed finite number of processes that are connected via communication media. Consider the following architecture and assume that communication is synchronous.

Consider a process D (the device), which communicates with two processes, P1 and P2, sending them regularly its current status (on or off). Every status message is forwarded, by P1 or P2, to L (a lamp, or display). The latter is either on or off and, upon reception of a message, may change (or not) the current status. For simplicity, we will assume rendez-vous (i.e., handshake) communication. A naive implementation looks as follows:
However, this may result in the following communication scenario, which yields the “wrong” output:

The problem is that $L$ does not know which of the last two messages that it receives is the latest one.

Suppose we are not allowed to change the architecture of the system. We are only allowed to add local and deterministic controllers to each process that add additional message contents. In particular, a controller should not block any of the possible actions of the given system. Is there still a way to control the protocol in a way such that $L$ shows the correct result?
1.1.2 A first solution.

A first, rather obvious solution would consist in using timestamps. Strictly increasing timestamps are sent by D along with a status message, forwarded by P1 and P2, 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.
1.1.3 Towards finite-state solutions

The previous solution uses infinitely many states, since time stamps are strictly increasing. This may not be adequate in the realm of reactive processes, which are supposed to run forever. However, finding a finite-state solution is difficult. It is actually impossible for the above specification.

Fortunately, there are guaranteed solutions in the case where every message is immediately followed by an acknowledgment. So, let us assume the following architecture:

Moreover, the allowed communication scenarios look as follows, i.e., every message is immediately followed by an acknowledgment (which can be augmented with additional messages):

In this particular case, we get a finite-state local controller. However, it is not always easy to come up with a solution.

Exercise 1.1. Try to find a finite-state controller.

The process of finding a controller is actually error-prone. So, it is natural to ask the following question:

Is there an automatic way to generate a controller from a specification?

The answer is YES, provided that the specification satisfies some properties.
In our example, the specification $L_{\text{spec}} \subseteq \Sigma^*$ could be a word language over the following alphabet:

$$
\Sigma = \{ \langle D \overset{on}{\rightarrow} P1 \rangle, \langle D \overset{on}{\rightarrow} P2 \rangle, \langle D \overset{off}{\rightarrow} P1 \rangle, \langle D \overset{off}{\rightarrow} P2 \rangle \}
\cup \{ \langle P1 \overset{on}{\rightarrow} L \rangle, \langle P2 \overset{on}{\rightarrow} L \rangle, \langle P1 \overset{off}{\rightarrow} L \rangle, \langle P2 \overset{off}{\rightarrow} L \rangle \}
\cup \{ \text{on}, \text{off}, \text{noop} \}
$$

Let $\Sigma_{P1} \subseteq \Sigma$ be the subalphabet containing those actions involving $P1$. Moreover, let $\Sigma_{P2} \subseteq \Sigma$ contain those with $P2$, and so on. In particular, $\{ \text{on}, \text{off}, \text{noop} \} \subseteq \Sigma_L$. With this, $L_{\text{spec}}$ is the set of words $w \in \Sigma^*$ satisfying the following:

(R1) The projection of $w$ to $\Sigma_{P1}$ is contained in

$$
\langle \langle D \overset{on}{\rightarrow} P1 \rangle \langle P1 \overset{on}{\rightarrow} L \rangle + \langle D \overset{off}{\rightarrow} P1 \rangle \langle P1 \overset{off}{\rightarrow} L \rangle \rangle^*.
$$

(R2) The projection of $w$ to $\Sigma_{P2}$ is contained in

$$
\langle \langle D \overset{on}{\rightarrow} P2 \rangle \langle P2 \overset{on}{\rightarrow} L \rangle + \langle D \overset{off}{\rightarrow} P2 \rangle \langle P2 \overset{off}{\rightarrow} L \rangle \rangle^*.
$$

(R3) The projection of $w$ to $\Sigma_L$ is contained in

$$
\left( \langle \langle P1 \overset{on}{\rightarrow} L \rangle + \langle P2 \overset{on}{\rightarrow} L \rangle + \langle P1 \overset{off}{\rightarrow} L \rangle + \langle P2 \overset{off}{\rightarrow} L \rangle \rangle (\text{on} + \text{off} + \text{noop}) \right)^*.
$$

(R4) “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 1.2.** Formalize the requirement (R4) in terms of a finite automaton or an MSO formula.  

Here are some example words to illustrate $L_{\text{spec}}$:

- $\langle D \overset{on}{\rightarrow} P1 \rangle \langle D \overset{off}{\rightarrow} P2 \rangle \langle P2 \overset{off}{\rightarrow} L \rangle \langle P1 \overset{off}{\rightarrow} L \rangle \text{noop} \in L_{\text{spec}}$,

- $\langle D \overset{on}{\rightarrow} P1 \rangle \langle P1 \overset{on}{\rightarrow} L \rangle \langle D \overset{on}{\rightarrow} P1 \rangle \text{on} \notin L_{\text{spec}}$,

- $\langle D \overset{on}{\rightarrow} P1 \rangle \langle P1 \overset{on}{\rightarrow} L \rangle \langle D \overset{off}{\rightarrow} P2 \rangle \langle P2 \overset{off}{\rightarrow} L \rangle \text{off} \in L_{\text{spec}}$,

- $\langle D \overset{on}{\rightarrow} P1 \rangle \langle P1 \overset{on}{\rightarrow} L \rangle \langle D \overset{off}{\rightarrow} P2 \rangle \langle P2 \overset{off}{\rightarrow} L \rangle \text{off} \in L_{\text{spec}}$.

Obviously, $L_{\text{spec}}$ is a regular language. Moreover, it is closed under permutation rewriting of independent events. The latter means that changing the order of neighboring independent actions in a word does not affect membership in $L_{\text{spec}}$. This seems natural, since the order of independent event cannot be enforced by a distributed protocol. Here, two actions are said to be independent if they involve distinct processes. For example,
• \( \langle D \overset{on}{\leftrightarrow} P1 \rangle \) and \( on \) are independent,

• \( \langle D \overset{on}{\leftrightarrow} P1 \rangle \) and \( \langle P2 \overset{on}{\leftrightarrow} L \rangle \) are independent,

• \( \langle D \overset{on}{\leftrightarrow} P1 \rangle \) and \( \langle D \overset{off}{\leftrightarrow} P2 \rangle \) are not independent,

• \( \langle D \overset{on}{\leftrightarrow} P2 \rangle \) and \( \langle P2 \overset{on}{\leftrightarrow} L \rangle \) are not independent.

**Exercise 1.3.** Show that \( L_{spec} \) is closed under permutation rewriting. ◊

The following is a fundamental result of concurrency theory (yet informally stated):

**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 \( L \).

Thus, the specification \( L_{spec} \) could indeed be realized as a distributed program.

There are, however, regular specifications that are not realizable (and, therefore, are not closed under permutation rewriting). Consider the language

\[
L = (\langle D \overset{on}{\leftrightarrow} P1 \rangle \langle P2 \overset{on}{\leftrightarrow} L \rangle)^* .
\]

Though \( L \) is regular, it is not closed under permutation rewriting. Even worse, the closure under permutation is not regular anymore. Specification \( L \) says that there are as many messages from \( D \) to \( P1 \) as from \( P2 \) to \( L \). Intuitively, it is clear that this cannot be realized by a finite-state system in a distributed fashion: There is no communication going on between \( D/P1 \) on the hand, and \( P2/L \) on the other hand.

### 1.2 Modeling behaviors as graphs

#### 1.2.1 Partial orders

Requirement (R4) from the last section is somehow awkward to write down. The reason is that a word over \( \Sigma \) imposes an ordering of events that, actually, are not causally related in the corresponding execution. When we say “last”, this refers to the “last” position in the word. Consider, for example, the word

\[
w = (\langle D \overset{on}{\leftrightarrow} P1 \rangle \langle P1 \overset{on}{\leftrightarrow} L \rangle \langle D \overset{off}{\leftrightarrow} P2 \rangle \overset{on}{\leftrightarrow} \langle D \overset{on}{\leftrightarrow} P1 \rangle \langle P1 \overset{on}{\leftrightarrow} L \rangle \langle P2 \overset{off}{\leftrightarrow} L \rangle \overset{noop}{\leftrightarrow} \in L_{spec} .
\]

The “last position” right before the \( on \) in the first line is actually in no way related to \( on \). So, it is not natural (and not needed) to include it in what we mean by
“last”. In our reasoning, a more relaxed ordering has to be recovered from the word ordering. So, why not directly reason about the causal order as it is imposed by a distributed system?

In the following, we do not consider an execution of a system as a word, i.e., a total order, but rather as a partial order. The partial order is already suggested by the message diagrams that we used to argue about our distributed protocols. Consider the execution below, whose partial order is represented by its Hasse diagram. It corresponds to the above word $w$.

Suppose this partial order is denoted by $\leq$. Then, (R4) can be conveniently rephrased (as we do not need the part “... that has already been followed by ...” anymore):

(R4’) “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.”

This can very easily be expressed in monadic second-order (MSO) logic over partial orders, using the partial order $\leq$.

**Exercise 1.4.** Give an MSO formula for (R4’). 

An advantage of MSO logic that is directly interpreted over partial orders is the following theorem:

<table>
<thead>
<tr>
<th>Theorem [Thomas 1990]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let $L$ be an MSO-definable set of partial orders. There is a finite-state distributed protocol that realizes $L$.</td>
</tr>
</tbody>
</table>
1.2.2 Reasoning about recursive processes

The previous discussion somehow motivates the naming of our lecture. However, in our modeling we will go a step further. One single behavior is not just a partial order, but an (acyclic) graph, which is more general. The edges reflect causal dependencies, but they provide even more information. For example, they may connect a procedure call with the corresponding return, or the sending of a message with its receive.

Consider a system of two processes, \( P_1 \) and \( P_2 \), connected by two unbounded FIFO channels (one in each direction). From time to time, \( P_1 \) sends requests to \( P_2 \). The latter, when receiving a request, calls a procedure, performs some internal actions, and returns from the procedure, before it sends an acknowledgment. In the scope of a procedure, it may call several subprocedures. Thus, \( P_1 \) performs actions from the alphabet \( \Sigma_1 = \{⟨!\text{req}⟩,⟨?\text{ack}⟩\} \) and \( P_2 \) performs actions from \( \Sigma_2 = \{⟨?\text{req}⟩,⟨!\text{ack}⟩,⟨\text{call}⟩,⟨\text{ret}⟩\} \). Let \( \Sigma = \Sigma_1 \cup \Sigma_2 \).

Let us try to model the protocol in terms of a word language \( L \). It should say that, whenever a request is received, \( P_2 \) should start a subroutine and send the acknowledgment immediately after returning from this subroutine. Thus, we shall have

\[
w_1 = ⟨!\text{req}⟩⟨?\text{req}⟩⟨\text{call}⟩⟨\text{call}⟩⟨\text{ret}⟩⟨\text{call}⟩⟨\text{call}⟩⟨\text{ret}⟩⟨\text{ack}⟩⟨?\text{ack}⟩ ∈ L.\]

On the other hand, we should have

\[
w_2 = ⟨!\text{req}⟩⟨?\text{req}⟩⟨\text{call}⟩⟨\text{call}⟩⟨\text{call}⟩⟨\text{call}⟩⟨\text{ret}⟩⟨\text{call}⟩⟨\text{call}⟩⟨\text{ret}⟩⟨\text{ack}⟩⟨?\text{ack}⟩ ⟨\text{ret}⟩ ∉ L.\]

But how do we express this in a specification language over words such as MSO logic? Unfortunately, there is no such formula, since \( L \) is a non-regular property.

Exercise 1.5. Prove that \( L \) is not regular.

The solution is to equip a word with additional information, which allows us to “jump” from a call to its associated return position. In other words, we add an edge from a call to the corresponding return:

Then, \( w_1 \) corresponds to:

Moreover, \( w_2 \) corresponds to:
From this point of view, we can now express our property easily in a suitable MSO logic:

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

It is for similar obvious reasons that, in an asynchronous setting, we connect a send with a receive event.

### 1.3 Underapproximate Verification

The lecture is concerned with distributed systems with a fixed architecture: There is a finite set of processes, which are connected via some communication media. We will consider stacks, FIFO channels, and bags (with the restriction that stacks connect a process with itself, with the purpose of modeling recursion).\(^1\) A priori, we assume that all media are unbounded. To get decidability or expressivity results, however, we may sometimes impose a bound on channels or stacks. Recall that, in the introductory example, we assumed synchronous communication, which roughly corresponds to FIFO channels with capacity 0.

The following figure illustrates one possible architecture (source: Aiswarya Cyriac’s thesis):

In the following, we will actually depict an architecture as a finite graph with directed edges of three types, depending on the type of the communication medium they represent. We will follow the following convention:

\(^1\)We do not consider lossy channels. Cf. Cours 2.9.1.
The most basic verification question is the *reachability problem*, i.e., to ask whether some control state of some process is reachable. Let us examine the decidability status of the reachability problem for the following architectures:

a) Undecidable. A system can simulate a Turing machine TM as follows: Via $c_1$, process 1 (on the left) sends the current configuration of the TM to process 2. Process 2 only sends every message that it receives immediately back to 1. When 1 receives the configuration, it locally modifies it simulating a transition of TM.

b) Undecidable. Similarly to a), the process sends the current configuration through the channel $c$. When receiving a configuration from $c$, it modifies it locally.

c) Undecidable. This can be shown by reduction from PCP (Post’s correspondence problem). Process 1 guesses a solution (a sequence of indices, and it sends the corresponding words via channels $c_1$ and $c_2$, respectively. Process 2 will then just check if the sequences send through $c_1$ and $c_2$ coincide. To do this, it reads alternately from both channels and checks whether both symbols coincide.

d) Decidable. This case can be reduced to synchronous communication and, therefore, reachability in a finite-state system.

e) Decidable. This case corresponds to emptiness of pushdown automata.

f) Undecidable. One can easily simulate a two-counter machine. Equivalently, we may use the concatenation of two stacks to simulate the unbounded tape of a Turing machine.
g) Undecidable. We may use a reduction from the intersection-emptiness problem for two pushdown automata. Modify the two pushdown automata (e.g., using Greibach’s normal form) such that they write nondeterministically some accepted word on the stack. So, both pushdown automata will first both choose words $w_1$ and $w_2$, respectively. While doing so, process 1 clears its stack sending $w_1$ letter by letter to process 2. Whenever process 2 receives a letter, it compares it with its stack content, which is then removed.

h) Undecidable. We use a reduction from case b). Process 1 simulates send transitions through channel $c$. To simulate a receive transition through $c$, it puts a token into bag $b_1$, whose value is the message to be received, say $m$. Process 1 can then only proceed when it finds an acknowledgment token in bag $b_2$. The latter is provided by process 2 after removing $m$ from $c$. Note that this procedure can be implemented even when communication between both processes is synchronous.

We conclude that almost all verification problems are undecidable even for very simple system architectures.

In this lecture, we therefore perform underapproximate verification: We restrict the behavior of a given system in a non-trivial way that still allows us to reason about it and deduce correctness/faultiness wrt. interesting properties. Let us illustrate some restrictions using some of the undecidable architectures above:

- In all cases, we may assume that communication media have capacity $B$ (existentially $B$-bounded), for some fixed $B$.
- In case f), assuming an order on the stacks, we can only pop from the first nonempty stack.
- In case f), we may also impose a bound on the number of contexts. In turn, there are several possible definitions of what is allowed in a context:
  - We can only touch one stack.
  - We can only pop from one stack.
  - Many more ...

Under all these restrictions, most standard verification problems (even model checking against MSO-definable properties) becomes decidable, with varying complexities.

In the lecture, we will take a uniform approach to underapproximate verification.
CHAPTER 2

Concurrent Processes with Data Structures

Notation is taken from [AG14].

2.1 The Model

Definition 2.1 (Architecture). An architecture is a tuple

\[ \mathfrak{A} = (\text{Procs}, \text{DS}, \text{Writer}, \text{Reader}) \]

- \text{Procs} finite set of processes
- \text{DS} = \text{Stacks} \sqcup \text{Queues} \sqcup \text{Bags} finite set of data structures
- \text{Writer} : \text{DS} \to \text{Procs}
- \text{Reader} : \text{DS} \to \text{Procs}

such that \( \text{Writer}(s) = \text{Reader}(s) \) for all \( s \in \text{Stacks} \).

Example 2.2. Consider the following architecture:

We have \( \text{Procs} = \{p_1, p_2\} \), \( \text{Stacks} = \{d_1\} \), \( \text{Queues} = \{d_2, d_3\} \), and \( \text{Bags} = \{d_4\} \).

E.g., \( \text{Writer}(d_1) = \text{Reader}(d_1) = p_1 \).
Definition 2.3 (CPDS). A system of concurrent processes with data structures (CPDS) over $\mathfrak{A}$ and an alphabet $\Sigma$ is a tuple

$$S = (\text{Locs}, \text{Val}, (\rightarrow_p)_{p \in \text{Procs}}, \ell_{\text{in}}, \text{Fin})$$

- Locs nonempty finite set of locations
- Val nonempty finite set of values
- $\ell_{\text{in}} \in \text{Locs}$ initial location
- $\text{Fin} \subseteq \text{Locs}^\text{Procs}$ global final locations
- $\rightarrow_p$ transitions of $p$
  - internal transition $\ell \xrightarrow{a} p \ell'$
  - write transition $\ell \xrightarrow{a,d,v} p \ell'$ with $\text{Writer}(d) = p$
  - read transition $\ell \xrightarrow{a,d,v} p \ell'$ with $\text{Reader}(d) = p$

where $\ell, \ell' \in \text{Locs}$, $a \in \Sigma$, $d \in \text{DS}$, and $v \in \text{Val}$

We let $\text{CPDS}(\mathfrak{A}, \Sigma)$ be the set of CPDSs over $\mathfrak{A}$ and $\Sigma$.

Example 2.4. Let $\mathfrak{A}$ be given by $\text{Procs} = \{p_1, p_2\}$, $\text{Queues} = \{c_1, c_2\}$, $\text{Stacks} = \{s\}$, $\text{Bags} = \emptyset$, with $\text{Writer}(c_1) = \text{Reader}(c_2) = p_1$ and $\text{Writer}(c_2) = \text{Reader}(c_1) = p_2$ and $\text{Writer}(s) = \text{Reader}(s) = p_2$. Moreover, let $\Sigma = \{a, b\}$. Consider the client-server system $\mathcal{S}_{cs}$ over $\mathfrak{A}$ and $\Sigma$ given as follows:
Process \(p_1\), the client, sends requests of type \(a\) or \(b\) to process \(p_2\), the server. The latter may acknowledge the request immediately, or put it on its stack (either, because it is busy or because the request does not have a high priority). At any time, however, the server may pop a task from the stack and acknowledge it.

We have \(\text{Locs} = \{0, 1, 2, 3, 4\}\), \(\ell_{\text{in}} = 0\), \(\text{Fin} = \{(0, 0)\}\), and \(\text{Val} = \{a, b\}\).

### 2.2 Operational Semantics

\(\mathcal{S}\) defines (infinite) automaton \(\mathcal{A}_\mathcal{S} = (\text{States}, \rightarrow, s_{\text{in}}, F)\) over \(\Gamma = (\text{Procs} \times \Sigma) \cup (\text{Procs} \times \Sigma \times \text{DS} \times \{!, ?\})\)

- **States** = \(\text{Locs}^{\text{Procs}} \times (\text{Val}^*)^\text{DS}\) for \((\ell, z) \in \text{States}\), we denote \(\ell = (\ell_p)_{p \in \text{Procs}}\) and \(z = (z_d)_{d \in \text{DS}}\)
- **\(s_{\text{in}}\)** = \(((\ell_{\text{in}}, \ldots, \ell_{\text{in}}), (\varepsilon, \ldots, \varepsilon))\)
- **\(F\)** = \(\text{Fin} \times \{\varepsilon\}^\text{DS}\)
- **\(\rightarrow \subseteq \text{States} \times \Gamma \times \text{States}\)**
  - internal transition \((\ell, z) \overset{p.a}{\rightarrow} (\ell', z')\) if \(\ell_p \xrightarrow{a} \ell'_p\) and \(\ell_q = \ell_q\) for all \(q \neq p\),
  - write transition \((\ell, z) \overset{p.a,d!}{\rightarrow} (\ell', z')\) if there is \(v \in \text{Val}\):
    \(\ell_p \xrightarrow{a,d!} \ell'_p\) and \(\ell'_q = \ell_q\) for all \(q \neq p\) and \(z'_d = z_d\) for all \(d \neq d\)
  - read transition \((\ell, z) \overset{p.a,d?}{\rightarrow} (\ell', z')\)
    if there is \(v \in \text{Val}\):
    \(\ell_p \xrightarrow{a,d?} \ell'_p\) and \(\ell'_q = \ell_q\) for all \(q \neq p\) and \(z'_d = z_d\) for all \(d \neq d\) and:
    \[
    \begin{cases}
      d \in \text{Stacks}: & z_d = uv \quad \text{and} \quad z'_d = u \\
      d \in \text{Queues}: & z_d = vw \quad \text{and} \quad z'_d = w \\
      d \in \text{Bags}: & z_d = uvw \quad \text{and} \quad z'_d = uw
    \end{cases}
    \]
    for some \(u, w \in \text{Val}^*\) and \(z'_{c} = z_{c}\) for all \(c \neq d\)

We let \(L_{\text{op}}(\mathcal{S}) := L(\mathcal{A}_\mathcal{S}) \subseteq \Gamma^*\) (discarding the empty word).

**Example 2.5.** In our client-server system, \(L_{\text{op}}(\mathcal{S}_{\text{cs}})\) contains:

- \((p_1, a, c_1!)(p_2, a, c_1?)(p_2, a, c_2!)(p_1, a, c_2?)\)
- \((p_1, a, c_1!)(p_1, b, c_1!)(p_2, a, c_1?)(p_2, a, s!)(p_2, b, c_1?)(p_2, b, c_2!)(p_1, b, c_2?)(p_2, a, s?)(p_2, a, c_2!)(p_1, a, c_2?)\)

**Exercise 2.6.** Show that \(L_{\text{op}}(\mathcal{S}_{\text{cs}})\) is not regular.
2.2.1 Nonemptiness/Reachability Checking

For an architecture $\mathcal{A}$ and an alphabet $\Sigma$, consider the following problem:

<table>
<thead>
<tr>
<th>NONEMPTINESS($\mathcal{A}, \Sigma$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance: $S \in \text{CPDS}(\mathcal{A}, \Sigma)$</td>
</tr>
<tr>
<td>Question: $L_{op}(S) \neq \emptyset$</td>
</tr>
</tbody>
</table>

**Theorem 2.7.** Let $\mathcal{A}$ be any of the following architectures: a, b, c, f, g, h. Then, NONEMPTINESS(\(\mathcal{A}, \Sigma\)) is undecidable.

The following table summarizes some special cases:
### Automata Types and Decidability

<table>
<thead>
<tr>
<th>( \mathfrak{A} )</th>
<th>automata type</th>
<th>CBM</th>
<th>Nonemptiness(( \mathfrak{A}, \Sigma ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>\text{Procs}</td>
<td>= 1)</td>
<td>finite automaton</td>
</tr>
<tr>
<td>(</td>
<td>\text{DS}</td>
<td>= 0)</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>\text{Procs}</td>
<td>=</td>
<td>\text{DS}</td>
</tr>
<tr>
<td>(</td>
<td>\text{Procs}</td>
<td>= 1)</td>
<td>multi-pushdown automaton</td>
</tr>
<tr>
<td>(</td>
<td>\text{DS}</td>
<td>\geq 2)</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>\text{DS}</td>
<td>= \text{Stacks})</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>\text{Procs}</td>
<td>\geq 2)</td>
<td>message-passing automaton</td>
</tr>
<tr>
<td>(</td>
<td>\text{DS}</td>
<td>= \text{Queues})</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>\text{Procs}</td>
<td>\times \text{Procs})</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>{ (p, p) \mid p \in \text{Procs} })</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Writer((p, q) = p)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reader((p, q) = q)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Graph Semantics

#### Example 2.8.

Let us represent behaviors as graphs. We start with an example.

![Graph Example]

16
Definition 2.9. A concurrent behavior with matching (CBM) over $\mathcal{A}$ and $\Sigma$ is a tuple

$$\mathcal{M} = ((w_p)_{p \in \text{Procs}}, (\triangleright^d)_{d \in \text{DS}})$$

- $w_p \in \Sigma^*$ sequence of actions on process $p$

**Notation:**
- $\mathcal{E}_p = \{(p, i) \mid 1 \leq i \leq |w_p|\}$ events of process $p$
- $\mathcal{E} = \bigcup_{p \in \text{Procs}} \mathcal{E}_p$
- $(p, i) \rightarrow (p, i + 1)$ if $1 \leq i < |w_p|$
- For $e = (p, i) \in \mathcal{E}_p$, let $\text{pid}(e) = p$ and $\lambda(e) \in \Sigma$ be the $i$-th letter of $w_p$.

- $\triangleright^d \subseteq \mathcal{E}_{\text{Writer}(d)} \times \mathcal{E}_{\text{Reader}(d)}$ such that:
  - if $e_1 \triangleright^d e_2$ and $e_3 \triangleright^d e_4$ are different edges ($d \neq d'$ or $(e_1, e_2) \neq (e_3, e_4)$), then they are disjoint ($|\{e_1, e_2, e_3, e_4\}| = 4$)
  - $\prec = (\rightarrow \cup \triangleright)^+ \subseteq \mathcal{E} \times \mathcal{E}$ is a strict partial order where $\triangleright = \bigcup_{d \in \text{DS}} \triangleright^d$
  - $\forall d \in \text{Stacks}$ (LIFO):
    - $e_1 \triangleright^d f_1$ and $e_2 \triangleright^d f_2$ and $e_1 < e_2 < f_1 \implies f_2 < f_1$
  - $\forall d \in \text{Queues}$ (FIFO):
    - $e_1 \triangleright^d f_1$ and $e_2 \triangleright^d f_2$ and $e_1 < e_2 \implies f_1 < f_2$

We let $\text{CBM}(\mathcal{A}, \Sigma)$ be the set of CBMs over $\mathcal{A}$ and $\Sigma$.

**Run:**
Consider a mapping $\rho : \mathcal{E} \rightarrow \text{Locs}$.

Define $\rho^- : \mathcal{E} \rightarrow \text{Locs}$ by

$$\rho^-(e) = \begin{cases} 
\rho(e') & \text{if } e' \rightarrow e \\
\ell_{\text{in}} & \text{if } e \text{ is minimal on its process}
\end{cases}$$

Now, $\rho$ is a run of $S$ on $\mathcal{M}$ if the following hold:

- for all internal events $e$: $\rho^-(e) \xrightarrow{\lambda(e)}_{\text{pid}(e)} \rho(e)$
- for all $e \triangleright^d f$, there is $v \in \text{Val}$ such that:
  - $\rho^-(e) \xrightarrow{\lambda(e), d, v}_{\text{pid}(e)} \rho(e)$ and
\[ - \rho^{-}(f) \xrightarrow{\lambda(f),d?w} \rho(f) \]

**Accepting:**

A run \( \rho \) is accepting if \((\ell_p)_{p \in \text{Procs}} \in \text{Fin}\) where

\[
\ell_p = \begin{cases} 
\ell_{\text{in}} & \text{if } \mathcal{E}_p = \emptyset \\
\rho((p, |w_p|)) & \text{otherwise}
\end{cases}
\]

We let \( L(S) \) denote the set of CBMs accepted by \( S \).

**Example 2.10.** The following figure depicts a run of \( S_{cs} \):

![Diagram of \( S_{cs} \) run]

**Relation between operational and graph semantics:**

Every CBM \( M = ((w_p)_{p \in \text{Procs}}, (\nabla^d)_d \in \text{DS}) \) defines a set of words over \( \Gamma \).

Let \( \gamma_M : \mathcal{E} \rightarrow \Gamma \quad (= (\text{Procs} \times \Sigma) \cup (\text{Procs} \times \Sigma \times \text{DS} \times \{!,?\})) \) be defined by

\[
\gamma_M(e) = \begin{cases} 
(pid(e), \lambda(e)) & \text{if } e \text{ is internal} \\
(pid(e), \lambda(e), d!) & \text{if } e \nabla^d f \\
(pid(e), \lambda(e), d?) & \text{if } f \nabla^d e
\end{cases}
\]
A linearization of $M$ is any (strict) total order $\sqsubseteq \subseteq E \times E$ such that $\leq \subseteq \sqsubseteq$. (recall that $\leq = (\to \cup \triangleright)^\dagger$).

Suppose $E = \{e_1, \ldots, e_n\}$ and $e_1 \sqsubseteq \ldots \sqsubseteq e_n$.

Then, $\sqsubseteq$ induces the word $\gamma_M(e_1) \ldots \gamma_M(e_n) \in \Gamma^*$.

Let $\text{Lin}(M) \subseteq \Gamma^*$ be the set of words that are induced by the linearisations of $M$.

**Remark 2.11.**
- If $\text{Bags} = \emptyset$, then for every $w \in \Gamma^*$, there is at most one $M \in \text{CBM}(A, \Sigma)$ such that $w \in \text{Lin}(M)$.
- If $\text{Bags} = \{d\}$, this is not the case: $(p, a, d!)(p, a, d!)(p, a, d?)(p, a, d?)$ is a linearization of two different CBMs.

**Example 2.12.** Let $M$ be the following CBM.

```
  p1
   \ a
   /\ b
  /   \ a
 p2
```

Then, $\text{Lin}(M)$ contains:

- $(p_1, a, c_1!)(p_1, b, c_1!)(p_2, a, s!)(p_2, b, c_1?)(p_2, b, c_2!)(p_2, a, s?)(p_2, a, c_2!)(p_1, b, c_2?)(p_1, a, c_2?)$
- $(p_1, a, c_1!)(p_2, a, c_1?)(p_1, b, c_1!)(p_1, b, c_2?)(p_2, a, s!)(p_2, b, c_1?)(p_2, b, c_2!)(p_2, a, s?)(p_2, a, c_2!)(p_1, b, c_2?)(p_1, a, c_2?)$

Actually, $M$ has 9 linearizations.

**Theorem 2.13.** For all $S \in \text{CPDS}(A, \Sigma)$, we have $\text{Lin}(L(S)) = L_{\text{op}}(S)$.

Without proof.
3.1 Monadic Second-Order Logic

Example: $\forall x (a(x) \Rightarrow \exists y (x \triangleright y \land b(y)))$

Syntax:
Let $\text{Var} = \{x, y, \ldots\}$ be an infinite set of first-order variables.
Let $\text{VAR} = \{X, Y, \ldots\}$ be an infinite set of second-order variables.
The set $\text{MSO}(\mathfrak{A}, \Sigma)$ of formulas from monadic second-order logic is given by the grammar:

$$\varphi ::= a(x) | p(x) | x = y | x \triangleright^d y | x \rightarrow y | x \in X | \varphi \lor \varphi | \neg \varphi | \exists x \varphi | \exists X \varphi$$

where $x, y \in \text{Var}$, $X \in \text{VAR}$, $a \in \Sigma$, $p \in \text{Procs}$, $d \in \text{DS}$.
The fragment $\text{EMSO}(\mathfrak{A}, \Sigma)$ consists of the formulas of the form $\exists X_1 \ldots \exists X_n \varphi$ where $\varphi$ is a first-order formula, i.e., it does not contain any second-order quantification.

Semantics:
Let $\mathcal{M} = ((w_p)_{p \in \text{Procs}}, (\triangleright^d)_{d \in \text{DS}})$ be a CBM. An $\mathcal{M}$-interpretation is a function $\mathcal{I}$ that maps every

- $x \in \text{Var}$ to some element of $\mathcal{E}$
- $X \in \text{VAR}$ to some subset of $\mathcal{E}$
Satisfaction $\mathcal{M} \models_I \varphi$ is defined inductively as follows:

- $\mathcal{M} \models_I a(x)$ if $\lambda(I(x)) = a$
- $\mathcal{M} \models_I p(x)$ if $\text{pid}(I(x)) = p$
- $\mathcal{M} \models_I x = y$ if $I(x) = I(y)$
- $\mathcal{M} \models_I x \triangledown d y$ if $I(x) \triangledown d I(y)$
- $\mathcal{M} \models_I x \rightarrow y$ if $I(x) \rightarrow I(y)$
- $\mathcal{M} \models_I x \in X$ if $I(x) \in I(X)$
- $\mathcal{M} \models_I \varphi \lor \psi$ if $\mathcal{M} \models_I \varphi$ or $\mathcal{M} \models_I \psi$
- $\mathcal{M} \models_I \neg \varphi$ if $\mathcal{M} \not\models_I \varphi$
- $\mathcal{M} \models_I \exists x \varphi$ if there is $e \in E$ such that $\mathcal{M} \models_{I[x \mapsto e]} \varphi$
- $\mathcal{M} \models_I \exists X \varphi$ if there is $E \subseteq E$ such that $\mathcal{M} \models_{I[X \mapsto E]} \varphi$

Here, $I[x \mapsto e]$ maps $x$ to $e$ and coincides with $I$ on $(\text{Var} \setminus \{x\}) \cup \text{VAR}$.

When $\varphi$ is a sentence, then $I$ is irrelevant, and we write $\mathcal{M} \models \varphi$ instead of $\mathcal{M} \models_I \varphi$.

We let $L(\varphi) := \{ \mathcal{M} \in \text{CBM}(\mathfrak{A}, \Sigma) | \mathcal{M} \models \varphi \}$.

---

**Example 3.1.** We use the following abbreviations:

- $\varphi \land \psi = \neg (\neg \varphi \lor \neg \psi)$
- $\forall x \varphi \equiv \neg \exists x \neg \varphi$  $\varphi \Rightarrow \psi \equiv \neg \varphi \lor \psi$
- $\text{write}(x) = \exists y (x \triangledown d y)$  $\text{read}(x) = \exists y (y \triangledown d x)$
- $\text{local}(x) = \neg \text{write}(x) \land \neg \text{read}(x)$
- $\text{min}(x) = \neg \exists y (y \rightarrow x)$  $\text{max}(x) = \neg \exists y (x \rightarrow y)$
- “$E_p = \emptyset$” $\equiv \neg \exists x p(x)$
- $x \triangledown y = \bigvee_{d \in \text{DS}} (x \triangledown_d y)$
- $x \leq y = \forall X (x \in X \land \forall z \forall z' ((z \in X \land (z \rightarrow z' \lor z \triangledown z'))) \Rightarrow z' \in X) \Rightarrow y \in X)$
- On CBMs, the latter formula is equivalent to

\[
\exists X [ x \in X \\
\land y \in X \\
\land \forall z \in X : \\
\exists z' \in X : z \rightarrow z' \lor z \triangledown z' ]
\]

\[\diamond\]
Example 3.2. We consider some formulas for $S_{cs}$:

- $\varphi_1 = \forall x (a(x) \Rightarrow \exists y (x \leq y \land b(y)))$

- $\text{req-ack}(x,y) = p_1(x) \land \left( \exists x_1, x_2 (x >^c_1 x_1 \rightarrow x_2 >^c_2 y) \lor \exists x_1, \ldots, x_4 (x >^c_1 x_1 \rightarrow x_2 >^s x_3 \rightarrow x_4 >^c_2 y) \right)$

- $\varphi_2 = \forall x, y \left( \text{req-ack}(x,y) \Rightarrow (((a(x) \land a(y)) \lor (b(x) \land b(y)))) \right)$

For the client-server system $S_{cs}$ from the previous chapter, we have $L(S_{cs}) \not\subseteq L(\varphi_1)$ and $L(S_{cs}) \subseteq L(\varphi_2)$.

3.2 Expressive Power of MSO Logic

Recall a theorem from the sequential case:

Theorem 3.3 (Büchi-Elgot-Trakhtenbrot [Büc60, Elg61, Tra62]).
Suppose $|\text{Procs}| = 1$ and $\text{DS} = \emptyset$. Let $L \subseteq \text{CBM}(\mathcal{A}, \Sigma)$, which can be seen as a word language $L \subseteq \Sigma^*$. Then, the following are equivalent:

- There is $S \in \text{CPDS}(\mathcal{A}, \Sigma)$ such that $L(S) = L$.
- There is a sentence $\varphi \in \text{MSO}(\mathcal{A}, \Sigma)$ such that $L(\varphi) = L$.

The theorem also holds for $|\text{Procs}| = 1$, $|\text{DS}| = 1$, and $\text{DS} = \text{Stacks}$ [AM09].

One direction is actually independent of architecture:

Theorem 3.4. For every $S \in \text{CPDS}(\mathcal{A}, \Sigma)$, there is a sentence $\varphi \in \text{EMSO}(\mathcal{A}, \Sigma)$ such that $L(\varphi) = L(S)$.
Proof. Fix $S = (\text{Locs}, \text{Val}, (\rightarrow_p)_{p \in \text{Procs}}, \ell_{\text{in}}, \text{Fin}) \in \text{CPDS}(\mathfrak{A}, \Sigma)$. We let

$$
\varphi = \exists (X_\ell)_{\ell \in \text{Locs}} \exists (Y_v)_v \in \text{Val} \left[ \begin{array}{l}
\text{Partition}((X_\ell)_{\ell \in \text{Locs}}) \\
\wedge \text{Partition}((Y_v)_v)
\wedge \forall x, y \bigwedge_{d \in \text{DS}} (x \triangleright^d y \Rightarrow \bigvee_{v \in \text{Val}} (x \in Y_v \land y \in Y_v))
\wedge \bigvee_{\ell = (\ell_p) \in \text{Fin}} \left( \bigwedge_{p \subseteq \text{Procs}} \forall p \in P, \ell_p = \ell_{\text{in}} \\
\left( \bigwedge_{p \in P} \mathcal{E}_p = \emptyset \\
\wedge \bigwedge_{p \in \text{Procs} \setminus P} \exists x (p(x) \land \max(x) \land x \in X_{\ell_p}) \right) \right)
\wedge \forall x, y \left( x \rightarrow y \land \text{local}(y) \Rightarrow \bigvee_{\ell \in \ell_{\text{in}}} \frac{a_{x,p}}{\rightarrow_p} \text{trans}_{\ell,a,p,e'}(x, y) \right)
\wedge \forall x \left( \min(x) \land \text{local}(x) \Rightarrow \bigvee_{\ell_{\text{in}}} a_{\rightarrow_p \ell} \text{trans}_{\ell,a,p,e}(x) \right)
\wedge \forall x, y \left( x \rightarrow y \land \text{write}(y) \Rightarrow \bigvee_{\ell_{\text{in}}} a_{d,v} \frac{a_{x,p}}{\rightarrow_p \ell} \left( \text{trans}_{\ell,a,p,e}(x, y) \land \text{data}_{d,v}(y) \right) \right)
\wedge \forall x \left( \min(x) \land \text{write}(x) \Rightarrow \bigvee_{\ell_{\text{in}}} a_{d,v} \frac{a_{x,p}}{\rightarrow_p \ell} \left( \text{trans}_{\ell,a,p,e}(x) \land \text{data}_{d,v}(x) \right) \right)
\wedge \forall x, y \left( x \rightarrow y \land \text{read}(y) \Rightarrow \bigvee_{\ell_{\text{in}}} a_{d,v} \frac{a_{x,p}}{\rightarrow_p \ell} \left( \text{trans}_{\ell,a,p,e}(x, y) \land \text{data}_{d,v}(y) \right) \right)
\wedge \forall x \left( \min(x) \land \text{read}(x) \Rightarrow \bigvee_{\ell_{\text{in}}} a_{d,v} \frac{a_{x,p}}{\rightarrow_p \ell} \left( \text{trans}_{\ell,a,p,e}(x) \land \text{data}_{d,v}(x) \right) \right) \right]
\right]
$$

where

- $\text{Partition}((X_\ell)_{\ell \in \text{Locs}}) = \forall x (\bigwedge_{\ell \in \ell_{\text{Locs}}} x \in X_\ell) \land \bigwedge_{\ell \neq \ell'} \exists x (x \in X_\ell \land x \in X_{\ell'})$

- $\text{trans}_{\ell,a,p,e}(x, y) = x \in X_\ell \land a(y) \land p(y) \land y \in X_{\ell'}$

- $\text{trans}_{\ell,a,p,e}(x) = a(x) \land p(x) \land x \in X_{\ell}$

- $\text{data}_{d,v}(x) = x \in Y_v \land \exists z (x \triangleright^d z \lor z \triangleright^d x)$

This completes the construction of the formula $\varphi$. We have $L(\varphi) = L(S)$. \hfill \blacksquare

Unfortunately, the other direction does not hold in general:

**Theorem 3.5.** Suppose $\Sigma = \{a, b, c\}$. Suppose that $\mathfrak{A}$ is given by $\text{Procs} = \{p_1, p_2\}$ and $\text{DS} = \text{Queues} = \{c_1, c_2\}$ with $\text{Writer}(c_1) = \text{Reader}(c_2) = p_1$ and $\text{Writer}(c_2) = \text{Reader}(c_1) = p_2$:

![Diagram](attachment:image.png)

There is a sentence $\varphi \in \text{MSO}(\mathfrak{A}, \Sigma)$ such that, for all $S \in \text{CPDS}(\mathfrak{A}, \Sigma)$, we have $L(S) \neq L(\varphi)$.
Proof. To illustrate the proof idea, which goes back to [Tho96] we consider pictures
over $\Sigma = \{a, b, c\}$. Here is an example:

Consider the set $P_\equiv$ of pictures that are of the form $ACA$ where

- $A$ is a nonempty square picture with labels in $\{a, b\}$, and
- $C$ is a $c$-labeled column.

The above picture is a member of $P_\equiv$.

The language $P_\equiv$ is definable by an MSO formula $\Phi_\equiv$ over pictures using predicates:

$$
go-right(x, y) \quad \text{go-down}(x, y)$$

The formula $\Phi_\equiv$ is easy to obtain once we have a “matching” predicate $\mu(x, y)$
that relates two coordinates $x$ and $y$ iff they refer to identical positions in the two
different square grids.

Essentially, $\mu(x, y)$ says that there are sets $X, Y$ and nodes $x_1, x_2, x_3$ such that

- $X$ contains all those nodes that are in the same row as $x$,
- $x_1$ is the topmost node on the same column as $x$,
- $x_2$ is on the intersection of the bottom row and the right-down diagonal
  starting from $x_1$,
- $x_3$ is reached from $x_2$ by going two steps to the right,
- $Y$ contains all those nodes that are in the same column as $x_3$.

Then, we have $\mu(x, y)$ iff $y \in X \cap Y$. The idea for $\mu(x, y)$ is illustrated below:
Next, we encode pictures into CBMs over $\mathcal{A}$ and $\Sigma$. The above picture is encoded as follows:

We obtain a formula $\tilde{\Phi}_\pi \in \text{MSO}(\mathcal{A}, \Sigma)$ for the encodings of the above picture language $P_\pi$ inductively:

- $\tilde{\exists x} \varphi = \exists x (\text{write}(x) \land \tilde{\varphi})$
- $\tilde{\text{go-right}}(x, y) = \exists z (x \triangleright z \rightarrow y)$
- $\tilde{\text{go-down}}(x, y) = \neg \text{bottom}(x) \land (x \rightarrow y \lor \exists z (x \rightarrow z \rightarrow y \land \neg \text{write}(z)))$
Here, \( \text{bottom}(x) \) says that \( x \) is an element that is located on the last row (left as an exercise). Other formulas remain unchanged.

Using Theorem 3.4, we can moreover determine a formula \( \psi_{\text{pict}} \) that describes the encodings of (arbitrary) pictures.

Let \( \varphi = \psi_{\text{pict}} \land \tilde{\Phi} = \sum \).

Towards a contradiction, suppose that there is \( S = (\text{Locs}, \text{Val}, (\rightarrow)^p_{p\in\text{Procs}}, \ell, \text{Fin}) \in \text{CPDS}(\mathcal{A}, \Sigma) \) such that \( L(S) = L(\varphi) \).

An accepting run of \( S \) has to transfer all the information it has about the upper part of the CBM along the middle part of size \( 2n \) (where \( n \) is the length of a column), to the lower part.

However, there are

- \( 2^{n^2} \) square pictures of width/height \( n \), and
- \( |\text{Locs}|^{2n} \)-many assignments of states to the middle part.

Thus, for sufficiently large \( n \), we can find an accepting run of \( S \) on a CBM \( M \) whose upper part and lower part do not match, i.e., \( M \notin L(\varphi) \).

However, there is a fragment of \( \text{MSO} \) that allows for a positive result (we do not present the proof).

**Theorem 3.6 ([BL06]).** Suppose \( DS = \text{Queues} \). Then, for every sentence \( \varphi \in \text{EMSO}(\mathcal{A}, \Sigma) \), there is a CPDS \( S \) such that \( L(S) = L(\varphi) \).

**Exercise 3.7.** Prove that CPDSs are, in general, not closed under complementation: Suppose \( \Sigma = \{a, b, c\} \) and assume the architecture \( \mathcal{A} \) from Theorem 3.5. Show that there is \( S \in \text{CPDS}(\mathcal{A}, \Sigma) \) such that, for all \( S' \in \text{CPDS}(\mathcal{A}, \Sigma) \), we have \( L(S') \neq \text{CBM}(\mathcal{A}, \Sigma) \setminus L(S) \).

**Exercise 3.8.** Show that Theorem 3.5 also holds when \( |\text{Procs}| = 1 \), \( |DS| = 2 \), and \( DS = \text{Stacks} \).

### 3.3 Propositional Dynamic Logic

Example:

\[
A(a \Rightarrow ((\rightarrow + \sum_{d \in DS} d^\ast) b)) \in \text{PDL}(\mathcal{A}, \Sigma)
\]

\[
\equiv \forall x(a(x) \Rightarrow \exists y(x \leq y \land b(y)))
\]
Syntax:
The syntax of \( \text{ICPDL}(\mathfrak{A}, \Sigma) \) is given by
\[
\Phi ::= E\sigma \mid \Phi \lor \Phi \mid \neg \Phi \\
\sigma ::= p \mid a \mid \sigma \lor \sigma \mid \neg \sigma \mid \langle \pi \rangle \sigma \\
\pi ::= \triangleright d \mid \rightarrow \mid \text{test}(\sigma) \mid \pi^{-1} \mid \pi + \pi \mid \pi \cap \pi \mid \pi \cdot \pi \mid \pi^*
\]
where \( p \in \text{Procs}, d \in \text{DS} \) and \( a \in \Sigma \). We call
- \( \sigma \) a state formula or node formula, and
- \( \pi \) a path formula.

If intersection \( \pi \cap \pi \) is not allowed, the fragment is PDL with converse (CPDL). If backward paths \( \pi^{-1} \) are not allowed the fragment is called PDL with intersection (IPDL). In simple PDL neither backwards paths nor intersection are allowed.

---

Let \( \mathcal{M} = ((w_p)_{p \in \text{Procs}}, (\triangleright d)_{d \in \text{DS}}) \in \text{CBM}(\mathfrak{A}, \Sigma) \).

Semantics of ICPDL formulas:
We write \( \mathcal{M} \models E\sigma \) if \( [\sigma]_\mathcal{M} \neq \emptyset \). Here, the semantics \( [\sigma]_\mathcal{M} \subseteq \mathcal{E} \) of state formulas is given below. For \( \Phi \in \text{ICPDL}(\mathfrak{A}, \Sigma) \), we let \( L(\Phi) := \{ \mathcal{M} \in \text{CBM}(\mathfrak{A}, \Sigma) \mid \mathcal{M} \models \Phi \} \).

Semantics of state formulas:
The semantics of a state formula \( \sigma \) wrt. \( \mathcal{M} \) is a set \( [\sigma]_\mathcal{M} \subseteq \mathcal{E} \), inductively defined as follows:
\[
[\langle \pi \rangle \sigma]_\mathcal{M} := \{ e \in \mathcal{E} \mid \text{there is } f \in [\sigma]_\mathcal{M} \text{ such that } (e, f) \in [\pi]_\mathcal{M} \}
\]
The relation \( [\pi]_\mathcal{M} \subseteq \mathcal{E} \times \mathcal{E} \) is defined below.
Semantics of path formulas:
The semantics of a path formula $\pi$ wrt. $M$ is a set $[[\pi]]_M \subseteq E \times E$, inductively defined as follows:

$$[[\triangleright^d]]_M := \triangleright^d \quad [[\rightarrow]]_M := \rightarrow$$

$$[[\text{test}(\sigma)]]_M := \{(e,e) \mid e \in [[\sigma]]_M\}$$

$$[[\pi^{-1}]]_M := [[\pi]]_M^{-1} = \{(f,e) \mid (e,f) \in [[\pi]]_M\}$$

$$[[\pi_1 + \pi_2]]_M := [[\pi_1]]_M \cup [[\pi_2]]_M \quad [[\pi_1 \cap \pi_2]]_M := [[\pi_1]]_M \cap [[\pi_2]]_M$$

$$[[\pi_1 \cdot \pi_2]]_M := [[\pi_1]]_M \circ [[\pi_2]]_M$$

$$[[\pi^*]]_M := [[\pi]]_M^* = \bigcup_{n \in \mathbb{N}} [[\pi]]_M^n$$

**Example 3.9.** Consider the following abbreviation/examples:

- $A\sigma = \neg E\neg \sigma$ (ICPDL formula)
- $[[\pi]]_\sigma = \neg \langle \pi \rangle \neg \sigma$ (state formula)
- true $= p \lor \neg p$ (state formula)
- $\triangleright = \sum_{d\in DS} \triangleright^d$ (path formula)
- write $= \langle \triangleright \rangle \text{true}$ (state formula)
- $\Phi_1 = A(a \Rightarrow \langle (\rightarrow + \triangleright)^* \rangle b)$ $\in$ PDL($\mathfrak{A}, \Sigma$)
- req-ack $= (\text{test}(p_1) \cdot \triangleright^{c_1} \cdot \rightarrow \cdot \triangleright^{c_2}) + (\text{test}(p_1) \cdot \triangleright^{c_1} \cdot \rightarrow \cdot \triangleright^* \cdot \rightarrow \cdot \triangleright^{c_2})$ (path formula; cf. client-server system from previous chapter)
- $\Phi_2 = A(a \Rightarrow [\text{req-ack}]a) \land A(b \Rightarrow [\text{req-ack}]b)$ $\in$ PDL($\mathfrak{A}, \Sigma$)
- $\equiv A([\text{test}(a) \cdot \text{req-ack}]a) \land A([\text{test}(b) \cdot \text{req-ack}]b)$ $\in$ PDL($\mathfrak{A}, \Sigma$)
- $\equiv \forall x,y \left( \text{req-ack}(x,y) \Rightarrow ((a(x) \land a(y)) \lor (b(x) \land b(y))) \right)$ $\in$ MSO($\mathfrak{A}, \Sigma$)

We have $L(S_{cs}) \not\subseteq L(\Phi_1)$ and $L(S_{cs}) \subseteq L(\Phi_2)$.  

\[\boxed{\diamond} \]
3.4 Expressive Power of ICPDL

One can show that ICPDL is no more expressive than MSO:

**Exercise 3.10.** Show that, for every formula $\Phi \in \text{ICPDL}(\mathfrak{A}, \Sigma)$, there is a sentence $\varphi \in \text{MSO}(\mathfrak{A}, \Sigma)$ such that $L(\varphi) = L(\Phi)$. \(\diamondsuit\)

Unfortunately, even IPDL is too expressive to be translatable into CPDSs:

**Theorem 3.11.** Suppose $\Sigma = \{a, b, c\}$ and let $\mathfrak{A}$ be given as follows:

There is $\Phi \in \text{IPDL}(\mathfrak{A}, \Sigma)$ such that, for all $\mathcal{S} \in \text{CPDS}(\mathfrak{A}, \Sigma)$, we have $L(\mathcal{S}) \neq L(\Phi)$.

**Exercise 3.12.** Prove Theorem 3.11 using the idea in the proof of Theorem 3.5.\(\diamondsuit\)

The exact relation between CPDL and CPDS is unknown. However, every PDL formula can be translated into a CPDS (the special case $\mathcal{DS} = \text{Queues}$ was considered in [BKM10]):

**Theorem 3.13 ([AGNK14]).** For every $\Phi \in \text{PDL}(\mathfrak{A}, \Sigma)$, there is $\mathcal{S} \in \text{CPDS}(\mathfrak{A}, \Sigma)$ such that $L(\mathcal{S}) = L(\Phi)$.

**Proof.** For a state formula $\sigma$, we construct, inductively,

$$S_\sigma = (\text{Locs}, \text{Val}, (\rightarrow p)_{p \in \text{Procs}}, \ell_{\text{in}}, \text{Fin}) \in \text{CPDS}(\mathfrak{A}, \Sigma)$$

together with a mapping $\gamma_\sigma : \text{Locs} \to \{0, 1\}$ such that

- $L(S_\sigma) = \text{CBM}(\mathfrak{A}, \Sigma)$ and,
- for all $\mathcal{M} \in \text{CBM}(\mathfrak{A}, \Sigma)$, all accepting runs $\rho$ of $S_\sigma$ on $\mathcal{M}$, and all events $e$ of $\mathcal{M}$, we have

  $$e \in [\sigma]_\mathcal{M} \text{ iff } \gamma_\sigma(\rho(e)) = 1.$$
CPDS $S_a$ for $a \in \Sigma$:
We let $S_a = (\text{Locs}, \text{Val}, (\rightarrow_p)_{p \in \text{Procs}}, \ell_{\text{in}}, \text{Fin})$ where

- $\text{Locs} = \{0, 1\}$
- $\text{Val} = \{v\}$
- $\ell_{\text{in}} = 1$
- $\text{Fin} = \{0, 1\}^{\text{Procs}}$

- transitions:
  
  - $\ell \overset{a}{\rightarrow} p_1$ and $\ell \overset{b}{\rightarrow} p_0$
  
  - $\ell \overset{a,d,v}{\rightarrow} p_1$ and $\ell \overset{b,d,v}{\rightarrow} p_0$ with $\text{Writer}(d) = p$
  
  - $\ell \overset{a,d,v}{\rightarrow} p_1$ and $\ell \overset{b,d,v}{\rightarrow} p_0$ with $\text{Reader}(d) = p$

for all $\ell \in \text{Locs}$, $b \neq a$, and $d \in \text{DS}$

Moreover, $\gamma_a(0) = 0$ and $\gamma_a(1) = 1$.

CPDS $S_p$ for $p \in \text{Procs}$:
We let $S_p = (\text{Locs}, \text{Val}, (\rightarrow_q)_{q \in \text{Procs}}, \ell_{\text{in}}, \text{Fin})$ where

- $\text{Locs} = \{0, 1\}$
- $\text{Val} = \{v\}$
- $\ell_{\text{in}} = 1$
- $\text{Fin} = \{0, 1\}^{\text{Procs}}$

- transitions:
  
  - $\ell \overset{a}{\rightarrow} p$
  
  - $\ell \overset{a,d,v}{\rightarrow} p_1$ with $\text{Writer}(d) = p$
  
  - $\ell \overset{a,d,v}{\rightarrow} q_1$ with $\text{Reader}(d) = p$
  
  - $\ell \overset{a}{\rightarrow} q_0$
  
  - $\ell \overset{a,d,v}{\rightarrow} q_0$ with $\text{Writer}(d) = q$
  
  - $\ell \overset{a,d,v}{\rightarrow} q_0$ with $\text{Reader}(d) = q$

for all $\ell \in \text{Locs}$, $a \in \Sigma$, $d \in \text{DS}$, and $q \neq p$

Again, we let $\gamma_p(0) = 0$ and $\gamma_p(1) = 1$.

CPDS $S_{\neg \sigma}$:
Suppose $S_\sigma = (\text{Locs}, \text{Val}, (\rightarrow_p)_{p \in \text{Procs}}, \ell_{\text{in}}, \text{Fin})$ with associated mapping $\gamma_\sigma : \text{Locs} \rightarrow \{0, 1\}$. We set $S_{\neg \sigma} = S_\sigma$ and let $\gamma_{\neg \sigma}(\ell) = 1 - \gamma_\sigma(\ell)$ for all $\ell \in \text{Locs}$.
Suppose $S_{\sigma_i} = (\text{Locs}_i, \text{Val}_i, (\neg \rightarrow_p)_{p \in \text{Procs}}, \ell_{in}^i, \text{Fin}_i)$, for $i \in \{1, 2\}$. We define $S_{\sigma_1 \lor \sigma_2} = (\text{Locs}, \text{Val}, (\neg \rightarrow_p)_{p \in \text{Procs}}, \ell_{in}, \text{Fin})$ as follows:

- $\text{Locs} = \text{Locs}_1 \times \text{Locs}_2$
- $\text{Val} = \text{Val}_1 \times \text{Val}_2$
- $\ell_{in} = (\ell_{in}^1, \ell_{in}^2)$
- $\text{Fin} = \{(\ell_{p \in \text{Procs}}^p, (\ell_{i \in \text{Procs}}^i)_{p \in \text{Procs}} | (\ell_{i \in \text{Procs}}^i)_{p \in \text{Procs}} \in \text{Fin}_i \text{ for all } i \in \{1, 2\}\}$
- transitions:
  - $(\ell_1, \ell_2) \xrightarrow{\alpha} (\ell_1', \ell_2')$ if $\ell_i \xrightarrow{\alpha} \ell_i'$ for all $i \in \{1, 2\}$
  - $(\ell_1, \ell_2) \xrightarrow{a.d\{v_1, v_2\}} (\ell_1', \ell_2')$ if $\ell_i \xrightarrow{a.d\{v_1, v_2\}} \ell_i'$ for all $i \in \{1, 2\}$
  - $(\ell_1, \ell_2) \xrightarrow{a.d\{v_1, v_2\}} (\ell_1', \ell_2')$ if $\ell_i \xrightarrow{a.d\{v_1, v_2\}} \ell_i'$ for all $i \in \{1, 2\}$

Finally, we let $\gamma_{\sigma}((\ell_1, \ell_2)) = \max\{\gamma_{\sigma_1}(\ell_1), \gamma_{\sigma_2}(\ell_2)\}$.

---

Let us turn to the case of formulas $\langle \pi \rangle_{\sigma}$. 4th Lecture

$\langle \pi \rangle_{\sigma} \equiv \langle \pi \cdot \text{test}(\sigma) \rangle_{\text{true}}$. Hence, we may assume that if $\langle \pi \rangle_{\sigma}$ appears as a subformula then $\sigma$ is true. Furthermore, we simply denote it by $\langle \pi \rangle$ (which means $\langle \pi \rangle_{\text{true}}$).

**Example 3.14.** Let us illustrate the idea by means of an example. Consider the PDL path formula

$$\pi = (\text{test}(a) \cdot (\rightarrow + \triangleright))^* \cdot \text{test}(b).$$

We translate $\pi$ into a finite automaton over the alphabet $\{\triangleright, \rightarrow, \text{test}(a), \text{test}(b)\}$ as follows:

![Finite automaton diagram]

The CPDS $S_{\langle \pi \rangle}$ will now label each event of a CBM with the set of states from which one “can reach a final state”, starting from the maximal events wrt. $<$ (in the example, the only $b$-labeled one):
This can indeed be achieved by a CPDS. To do so, the CPDS has to “inspect”, at each event \( e \), the states at the immediate \( \to \) - and \( \triangleright^d \)-successors of \( e \). In particular, at a write event, it will have to guess what will be the state at the corresponding read event. Finally, an event satisfies \( \langle \pi \rangle \) iff the initial state \( 0 \) is contained in the labeling.

\[\text{CPDS } S(\pi) :\]

Let \( \text{Tests}(\pi) = \{\text{test}(\sigma_1), \ldots, \text{test}(\sigma_n)\} \) be the set of tests appearing in \( \pi \).

Now, \( \pi \) can be seen as a regular expression over the alphabet

\[\Omega = \text{Tests}(\pi) \cup \{\triangleright^d \mid d \in DS\} \cup \{\to\}.\]

Let \( B = (S, \delta, \iota, F) \) be a finite automaton over \( \Omega \) for \( \pi \), i.e., such that \( L(B) = L(\pi) \subseteq \Omega^* \). Note that we can assume \( |S| = |\pi| \). Given \( s \in S \), we set \( B_s = (S, \delta, s, F) \), i.e., \( B_s \) is essentially \( B \), but with new initial state \( s \).

Let \( \pi_s \) be a rational expression over \( \Omega \) that is equivalent to \( B_s \) (in particular, \( \pi_i = \pi \)).

For a CBM \( \mathcal{M} = ((w_p)_{p \in \text{Procs}}, (\triangleright^d_{d \in \text{DS}})) \), we want to compute \( \nu : \mathcal{E} \to 2^S \) such that

\[\nu(e) = \{s \in S \mid e \in [\langle \pi_s \rangle]_{\mathcal{M}}\}.\]

Let \( H(e) = \{\text{test}(\sigma_i) \mid e \in [\sigma_i]_{\mathcal{M}}\} \subseteq \Omega \).

For \( K \subseteq \Omega^* \) and \( T \subseteq S \), let

\[\delta^{-1}(K,T) = \{s \in S \mid \delta(s,w) \in T \text{ for some } w \in K\}.\]

Let \( \nu^+(e) = \begin{cases} \nu(f) & \text{if } e \to f \\ \emptyset & \text{if } e \text{ is maximal on its process.} \end{cases} \)

**Lemma 3.15.** (i) If \( e \) is not a write event, then

\[\nu(e) = \delta^{-1}(H(e)^*, F) \cup \delta^{-1}(H(e)^* \to, \nu^+(e))\].
(ii) If $e \triangleright^d f$, then
\[
\nu(e) = \delta^{-1}(H(e)^*, F) \cup \delta^{-1}(H(e)^* \rightarrow, \nu^+(e)) \cup \delta^{-1}(H(e)^* \triangleright^d, \nu(f)).
\]


Remark 3.17. If $e$ is maximal wrt. $<$, then $\nu(e) = \delta^{-1}(H(e)^*, F)$ can be computed directly.

We are looking for a CPDS that computes $\nu$.

Problem: The computation of $\nu$ goes backward, whereas a CPDS run goes forward.

Solution: Guess $\nu$ nondeterministically and check afterwards whether the guess was correct.

We define $S_{\{\pi\}} = S_{\sigma_1} \times \ldots \times S_{\sigma_n} \times S_B$. By induction, the $(S_{\sigma_i})_{1 \leq i \leq n}$ are given.

From accepting runs $(\rho_i)_{1 \leq i \leq n}$ of $(S_{\sigma_i})_{1 \leq i \leq n}$, we get
\[
H(e) = \{\text{test}(\sigma_i) \mid \gamma_{\sigma_i}(\rho_i(e)) = 1\},
\]
which, by induction, equals $\{\text{test}(\sigma_i) \mid e \in [\sigma_i]_{M}\}$.

We define $S_B = (\text{Locs}, \text{Val}, (\rightarrow_p)_{p \in \text{Procs}}, \text{Init}, \text{Fin})$ as follows:

- **Locs** = $2^S \times 2^S$
  In $(U, V) \in \text{Locs}$, $U$ represents $\nu(e)$, and $V$ represents the “guessed” $\nu^+(e)$.

- **Val** = $2^S$
  Here, $U \in \text{Val}$ represents $\nu(f)$ when $e \triangleright^d f$.

- **Init** = $\{(\emptyset) \times 2^S\}_{\text{Procs}} = \{((\emptyset, V) \mid V \subseteq S\}_{\text{Procs}}$
  We assume a set of global initial states, which is a generalization compared with single local initial states, but can be simulated with nondeterminism.

- **Fin** = $(2^S \times \{\emptyset\})_{\text{Procs}} = \{(U, \emptyset) \mid U \subseteq S\}_{\text{Procs}}$

Now, we turn to the transitions.

In the following, let
\[
\Theta(G, V) = \delta^{-1}(G^*, F) \cup \delta^{-1}(G^* \rightarrow, V) \text{ and }
\Psi_d(G, V, W) = \Theta(G, V) \cup \delta^{-1}(G^* \triangleright^d, W).
\]
• internal:
\[(U', U) \xrightarrow{G_p} (U, V) \quad \text{if } U = \Theta(G, V)\]

• write \((\text{Writer}(d) = p)\):
\[(U', U) \xrightarrow{G,W_d} (U, V) \quad \text{if } U = \Psi_d(G, V, W)\]

• read \((\text{Reader}(d) = p)\):
\[(U', U) \xrightarrow{G,U_d} (U, V) \quad \text{if } U = \Theta(G, V)\]

**Remark 3.18.** If we see the CPDS working backward, then it is deterministic (forward modalities = backward deterministic).

Finally, we set
\[\gamma_{\langle \pi \rangle}(U, V) = \begin{cases} 1 & \text{if } \iota \in U \\ 0 & \text{otherwise} \end{cases}\]
and \(\gamma_{\langle \pi \rangle}(\ell_1, \ldots, \ell_n, U, V) = \gamma_{B}(U, V)\).

The correctness proof for \(S_{\langle \pi \rangle}\) is by induction on the partial order induced by a CBM, starting from the maximal events.

It remains to define automata for PDL formulas \(\Phi \in \text{PDL}(\mathfrak{A}, \Sigma)\). Without loss of generality, we assume that \(\Phi\) is a positive boolean combination of formulas of the form \(E\sigma\) or \(A\sigma\). Disjunction and conjunction are easy to handle, since CPDSs are closed under union and intersection (exercise).

For the case \(A\sigma\), suppose we already have \(S_{\sigma} = (\text{Locs}, \text{Val}, (\neg p)_{p \in \text{Procs}}, \ell_{\text{in}}, \text{Fin})\) with associated mapping \(\gamma_{\sigma} : \text{Locs} \to \{0, 1\}\). Without loss of generality, we assume \(\gamma_{\sigma}(\ell_{\text{in}}) = 1\). Then, the CPDS for \(A\sigma\) is simply the restriction of \(S\) to those locations \(\ell\) such that \(\gamma_{\sigma}(\ell) = 1\).

The case \(E\sigma\) is left as an exercise.

**Exercise 3.19.** Consider the following extension of \(\text{PDL}(\mathfrak{A}, \Sigma)\), which we call \(\text{PDL}^{-1}(\mathfrak{A}, \Sigma)\):

\[
\Phi ::= E\sigma \mid \Phi \lor \Phi \mid \neg \Phi \\
\sigma ::= p \mid a \mid \sigma \lor \sigma \mid \neg \sigma \mid \langle \pi \rangle \sigma \mid \langle \pi^{-1} \rangle \sigma \\
\pi ::= \triangleright^d \mid \rightarrow \mid \text{test}(\sigma) \mid \pi + \pi \mid \pi \cdot \pi \mid \pi^* 
\]

where \(p \in \text{Procs}, d \in \text{DS}\) and \(a \in \Sigma\). Show that Theorem 3.13 even holds for the logic \(\text{PDL}^{-1}(\mathfrak{A}, \Sigma)\):

For every \(\Phi \in \text{PDL}^{-1}(\mathfrak{A}, \Sigma)\), there is \(S \in \text{CPDS}(\mathfrak{A}, \Sigma)\) such that \(L(S) = L(\Phi)\).
Since PDL is closed under complementation (negation), while CPDSs are not (for certain architectures), we obtain, as a corollary, that CPDSs are strictly more expressive than PDL.

3.5 Satisfiability and Model Checking

For an architecture $\mathfrak{A}$ and an alphabet $\Sigma$, consider the following problems:

<table>
<thead>
<tr>
<th>MSO-Satisfiability($\mathfrak{A}, \Sigma$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance: $\varphi \in \text{MSO}(\mathfrak{A}, \Sigma)$</td>
</tr>
<tr>
<td>Question: $L(\varphi) \neq \emptyset$ ?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(IC)PDL-Satisfiability($\mathfrak{A}, \Sigma$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance: $\Phi \in (\text{IC})\text{PDL}(\mathfrak{A}, \Sigma)$</td>
</tr>
<tr>
<td>Question: $L(\Phi) \neq \emptyset$ ?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSO-ModelChecking($\mathfrak{A}, \Sigma$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance: $S \in \text{CPDS}(\mathfrak{A}, \Sigma) ; \varphi \in \text{MSO}(\mathfrak{A}, \Sigma)$</td>
</tr>
<tr>
<td>Question: $L(S) \subseteq L(\varphi)$ ?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(IC)PDL-ModelChecking($\mathfrak{A}, \Sigma$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance: $S \in \text{CPDS}(\mathfrak{A}, \Sigma) ; \Phi \in (\text{IC})\text{PDL}(\mathfrak{A}, \Sigma)$</td>
</tr>
<tr>
<td>Question: $L(S) \subseteq L(\Phi)$ ?</td>
</tr>
</tbody>
</table>

**Theorem 3.20.** Let $\mathfrak{A}$ be given as follows (and $\Sigma$ be arbitrary):

\[ \text{P}_1 \xrightarrow{e_1} \text{P}_2 \xrightarrow{e_2} \text{P}_1 \]

Then, all the abovementioned problems are undecidable.

By Theorem 3.13, we obtain the following positive result:
Theorem 3.21. Suppose $DS = \text{Bags}$. Then, the problems

\begin{align*}
\text{PDL-Satisfiability}(\mathfrak{A}, \Sigma) \quad \text{and} \\
\text{PDL-ModelChecking}(\mathfrak{A}, \Sigma)
\end{align*}

are both decidable.
CHAPTER 4

Underapproximate Verification

Recall that most verification problems such as nonemptiness, global-state reachability, and model checking are undecidable even for very simple architectures.

4.1 Principles of Underapproximate Verification

To get decidability, we will restrict decision problems to a subclass \( \mathcal{C} \subseteq \text{CBM}(\mathcal{A}, \Sigma) \) of CBMs:

<table>
<thead>
<tr>
<th>MSO-VALIDITY(( \mathcal{A}, \Sigma, \mathcal{C} ))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance:</strong> ( \varphi \in \text{MSO}(\mathcal{A}, \Sigma) )</td>
</tr>
<tr>
<td><strong>Question:</strong> ( \mathcal{C} \subseteq L(\varphi) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSO-MODELCHECKING(( \mathcal{A}, \Sigma, \mathcal{C} ))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance:</strong> ( S \in \text{CPDS}(\mathcal{A}, \Sigma); \varphi \in \text{MSO}(\mathcal{A}, \Sigma) )</td>
</tr>
<tr>
<td><strong>Question:</strong> ( L(S) \cap \mathcal{C} \subseteq L(\varphi) )</td>
</tr>
</tbody>
</table>

For example, we may only consider the CBMs that can be executed when the data structures have bounded capacity:

**Definition 4.1.** Let \( k \geq 0 \). A CBM \( \mathcal{M} \) is called \( k \)-existentially bounded (\( k\text{-}\exists B \) for short) if there is a linearization \( w \in \text{Lin}(\mathcal{M}) \) such that, for every prefix \( u \) of \( w \), the number of unmatched writes in \( u \) is at most \( k \). A class \( \mathcal{C} \subseteq \text{CBM}(\mathcal{A}, \Sigma) \) is \( k\text{-}\exists B \) if \( \mathcal{M} \) is \( k\text{-}\exists B \) for every \( \mathcal{M} \in \mathcal{C} \). Finally, \( \mathcal{C} \) is called \( \exists B \) if it is \( k\text{-}\exists B \) for some \( k \). \( \diamond \)
**Example 4.2.** We will give some examples:

(a) The CBM from Example 2.8 is $3
\exists B$.

(b) The class of nested words ($|\text{Procs}| = 1$, $|\text{DS}| = 1$, and $\text{DS} = \text{Stacks}$) is not $\exists B$, as illustrated by the following figure:

![Diagram](image)

(c) The class of MSCs ($|\text{Procs}| \geq 2$, $\text{DS} = \text{Queues} = \text{Procs} \times \text{Procs} \setminus \{(p,p) \mid p \in \text{Procs}\}$, $\text{Writer}(p,q) = p$, and $\text{Reader}(p,q) = q$) is not $\exists B$:

![Diagram](image)

**Exercise 4.3.** Consider the encoding of pictures as CBMs from Section 3.2. Show that the encoding of a picture of height $k$ yields a CBM that is $k-\exists B$.

We are looking for “reasonable” classes of CBMs that are suitable for underapproximate verification.

**Definition 4.4.** Let $\mathcal{C} = (C_k)_{k \geq 0}$ with $C_k \subseteq \text{CBM}(\mathfrak{A}, \Sigma)$ be a family of classes of CBMs. Then, $\mathcal{C}$ is called

- monotone if $C_k \subseteq C_{k+1}$ for all $k \geq 0$,
- complete if $\bigcup_{k \geq 0} C_k = \text{CBM}(\mathfrak{A}, \Sigma)$,
- decidable if the usual decision problems are decidable when the domain of CBMs is restricted to $C_k$,
- MSO-definable if, for all $k \geq 0$, there is a sentence $\varphi_k \in \text{MSO}(\mathfrak{A}, \Sigma)$ such that $L(\varphi_k) = C_k$, and
• CPDS-definable if, for all \( k \geq 0 \), there is a CPDS \( S_k \in \text{CPDS}(\mathfrak{A}, \Sigma) \) such that \( L(S_k) = C_k \).

We will see several concrete examples of such families in Section ??.

Below, we first present a generic family, which is based on the notion of *split-width*.

### 4.2 Graph-Theoretic Approach

In the following, we will use tools from graph theory. Actually, the pair \((\mathfrak{A}, \Sigma)\)
defines a *signature* of unary and binary relation symbols. Thus, one can consider
general graphs over \((\mathfrak{A}, \Sigma)\), with node labels from \(\Sigma\) and edge labels from \(\{\text{proc}\} \cup DS\). Here, \(\text{proc}\) stands for *process successor*, and \(d \in DS\) is the labeling of an edge
that connects a write and a read event. Those graphs that satisfy the axioms from
Definition 2.9 can then be considered as CBMs.

**Proposition 4.5.** The class \(\text{CBM}(\mathfrak{A}, \Sigma)\) is MSO-definable, i.e., there is a sentence \(\varphi_{\text{cbm}} \in \text{MSO}(\mathfrak{A}, \Sigma)\) such that, for all graphs \(G\) over \((\mathfrak{A}, \Sigma)\), we have

\[
G \models \varphi_{\text{cbm}} \iff G \text{ is a CBM over } (\mathfrak{A}, \Sigma).
\]

**Proposition 4.6.** Fix a class \(C \subseteq \text{CBM}(\mathfrak{A}, \Sigma)\). The following problems are inter-
reducible:

1. MSO-VALIDITY(\(\mathfrak{A}, \Sigma, C\))
2. MSO-MODEL-CHECKING(\(\mathfrak{A}, \Sigma, C\))

*Proof.* For the reduction from 1. to 2., let \(\varphi \in \text{MSO}(\mathfrak{A}, \Sigma)\) be a sentence. Let \(S_{\text{univ}}\) be the universal CPDS, satisfying \(L(S_{\text{univ}}) = \text{CBM}(\mathfrak{A}, \Sigma)\). Note that we can define \(S_{\text{univ}}\) such that \(|\text{Locs}| = 1 = |\text{Val}|\) and where we have full transition tables. We have:

\[
C \subseteq L(\varphi) \iff L(S_{\text{univ}}) \cap C \subseteq L(\varphi)
\]

For the reduction from 2. to 1., let \(S \in \text{CPDS}(\mathfrak{A}, \Sigma)\) and \(\varphi \in \text{MSO}(\mathfrak{A}, \Sigma)\). Let \(\varphi_S \in \text{MSO}(\mathfrak{A}, \Sigma)\) such that \(L(\varphi_S) = L(S)\) (cf. Theorem 3.4). We have:

\[
L(S) \cap C \subseteq L(\varphi) \\
\quad \iff L(\varphi_S) \cap C \subseteq L(\varphi) \\
\quad \iff C \subseteq L(\varphi \lor \neg \varphi_S)
\]

The decidability of the MSO theory of classes of graphs has been extensively studied
(cf. the book by Courcelle and Engelfriet [CE12]):

**Theorem 4.7.** Let \(C\) be a class of bounded degree graphs which is MSO-definable. The following statements are equivalent:

1. \(C\) has a decidable MSO theory.
2. \( \mathcal{C} \) can be interpreted in binary trees.
3. \( \mathcal{C} \) has bounded tree-width.
4. \( \mathcal{C} \) has bounded clique-width.
5. \( \mathcal{C} \) has bounded split-width\(^1\) (if \( \mathcal{C} \) is a class of CBMs).

For a class \( \mathcal{C} \subseteq \text{CBM}(\mathcal{A}, \Sigma) \) that is MSO-definable, we prove bounded split-width

- to get decidability,
- to get the interpretation in binary trees,
- to reduce verification problems to problems on tree automata, and
- to get efficient algorithms with optimal complexity.

In the theorem above, graphs are interpreted in binary trees. We need to identify which trees are valid encodings, i.e., do encode graphs in the class \( \mathcal{C} \). This is why we assumed the class of graphs to be MSO-definable. From this, we can build a tree automaton for the valid encodings. Actually, we can replace MSO-definability of the class \( \mathcal{C} \) by the existence of a tree automaton \( \mathcal{A}_\mathcal{C} \) for the valid encodings of CBMs in \( \mathcal{C} \). It is often better to define the tree automaton directly. Its size has a direct impact on the decision procedures arising from the tree-interpretation.

### 4.3 Graph (De)composition and Tree Interpretation

We will illustrate the concept of tree interpretation by means of cographs. Undirected and labeled cographs are generated by cograph terms. A cograph term is built from the grammar (cograph algebra)

\[
C ::= a \mid C \oplus C \mid C \otimes C
\]

where \( a \in \Sigma \). A term \( C \) defines a cograph \([C] = (V, E, \lambda)\) as follows:

- \([a]\) is the graph \((\{1\}, \emptyset, 1 \mapsto a)\) with one \(a\)-labeled vertex and no edges
- if \([C_i] = (V_i, E_i, \lambda_i)\) for \(i = 1, 2\), with \(V_1 \cap V_2 = \emptyset\), then
  \[
  [C_1 \oplus C_2] = (V_1 \cup V_2, E_1 \cup E_2, \lambda_1 \cup \lambda_2)
  \]
- if \([C_i] = (V_i, E_i, \lambda_i)\) for \(i = 1, 2\), with \(V_1 \cap V_2 = \emptyset\), then
  \[
  [C_1 \otimes C_2] = (V_1 \cup V_2, E_1 \cup E_2 \cup (V_1 \times V_2) \cup (V_2 \times V_1), \lambda_1 \cup \lambda_2)
  \]
  (this is called the complete join)

\(^1\)Split-width will be defined below.
**Example 4.8.** Consider the cograph term \( C = ((a \otimes a) \otimes b) \otimes (a \oplus (a \otimes b)) \). The figure below shows a tree representation of \( C \) as well as the cograph \([C]\) defined by \( C \).

![Diagram](image)

Note that the tree representation offers a top-down decomposition of a cograph. ♦

**Remark 4.9.** One can also include edge labelings \( d \), using operators \( \otimes_d \) with the expected meaning.

Clearly, the set of cograph terms (considered as trees) is a regular tree language:

**Lemma 4.10.** The set of cograph terms is MSO-definable in binary trees. Alternatively, there is a tree automaton which accepts the set of binary trees that are cograph terms.

**Proof.** The set of “valid” binary trees is defined by the sentence

\[
\varphi_{\text{cograph}} = \forall x \left( \text{leaf}(x) \rightarrow \bigvee_{a \in \Sigma} a(x) \right) \land \neg \text{leaf}(x) \rightarrow \bigoplus(x) \lor \bigotimes(x).
\]

**Tree interpretation:**

Next, we demonstrate that a cograph can be recovered from its cograph term using MSO formulas, i.e., an MSO interpretation. Let \( C \) be a cograph term (i.e., a binary tree), and let \( G = [C] = (V, E, \lambda) \).

- The nodes in \( V \) correspond to the leaves of \( C \):

\[
\varphi_{\text{vertex}}(x) = \text{leaf}(x) = \neg \exists y (x \downarrow_0 y \lor x \downarrow_1 y)
\]

where \( \downarrow_0 \) stands for “left child” and \( \downarrow_1 \) for “right child”.

- The set \( E \) of edges is defined by

\[
\varphi_{\text{edge}}(x, y) = \text{"least-common-ancestor}(x, y) \text{ is labeled } \otimes\}
\]

\[
= \exists z, x', y' \left( \begin{array}{c}
\text{z } \downarrow_0 x' \land z \downarrow_1 y' \\
\lor \text{ z } \downarrow_1 x' \land z \downarrow_0 y'
\end{array} \right) \land (x' \downarrow^* x) \land (y' \downarrow^* y) \land \otimes(z)
\]

41
Proposition 4.11. MSO formulas over cographs can be “translated” to MSO formulas over cograph terms: For all sentences $\varphi \in \text{MSO}(\Sigma, E)$, there is a sentence $\bar{\varphi} \in \text{MSO}(\Sigma \cup \{\oplus, \otimes\}, \downarrow_0, \downarrow_1)$ such that, for every cograph term $C$, say with $G = \llbracket C \rrbracket$, we have

$$G \models \varphi \iff C \models \bar{\varphi}.$$ 

Proof. We proceed by induction on $\varphi$:

- $\bar{a(x)} = a(x)$
- $\bar{x E y} = \varphi_{edge}(x, y)$
- $\bar{x \in X} = x \in X$
- $\bar{x = y} = (x = y)$
- $\bar{\neg \varphi} = \neg \bar{\varphi}$
- $\bar{\varphi_1 \lor \varphi_2} = \bar{\varphi_1} \lor \bar{\varphi_2}$
- $\bar{\exists x \varphi} = \exists x(\varphi_{vertex}(x) \land \bar{\varphi})$
- $\bar{\exists X \varphi} = \exists X((\forall x(x \in X \rightarrow \varphi_{vertex}(x))) \land \bar{\varphi})$

Note that, in the correctness proof, we have to deal with free variables. Actually, the inductive statement is as follows: For all $\varphi \in \text{MSO}(\Sigma, E)$, there is $\bar{\varphi} \in \text{MSO}(\Sigma \cup \{\oplus, \otimes\}, \downarrow_0, \downarrow_1)$ such that, for every cograph term $C$ and every interpretation $I$ of $\text{Var} \cup \text{VAR}$ in $\text{Leaves}(C) = \text{Vertices}(G = \llbracket C \rrbracket)$, we have $G \models_I \varphi$ iff $C \models_I \bar{\varphi}$.

Corollary 4.12. The MSO theory of cographs is decidable.

Proof. Let $\varphi \in \text{MSO}(\Sigma, E)$. Then, $\varphi$ is valid on cographs iff $\varphi_{\text{cograph}} \rightarrow \bar{\varphi}$ is valid on binary trees (cf. Lemma 4.10). Note that, moreover, $\varphi$ is satisfiable on cographs iff $\varphi_{\text{cograph}} \land \bar{\varphi}$ is satisfiable on binary trees. The result from the corollary follows, since MSO validity is decidable on binary trees. Indeed, the problem can be reduced to tree-automata emptiness [TW68].
4.4 Split-Width

Next, we give a short account of split-width. See also Section 6 of [AG14]. We start with some definitions. A split-CBM is a CBM in which behaviors of processes may be split in several factors.

**Definition 4.13.** A graph \( \mathcal{M} = (\mathcal{E}, \rightarrow, (\triangleright^d)_{d \in DS}, \text{pid}, \lambda) \) is a split-CBM if it is possible to obtain a CBM \( (\mathcal{E}, \rightarrow \cup \triangleright, (\triangleright^d)_{d \in DS}, \text{pid}, \lambda) \) by adding some missing process edges \( \triangleright \in \mathcal{E}^2 \). Missing edges are called splits (or holes). A factor (or block) of \( \mathcal{M} \) is a maximal sequence of events connected by process edges. The width\(^2\) of \( \mathcal{M} \) is its number of factors. It is denoted \( \text{width}(\mathcal{M}) \).

We say that \( \mathcal{M} \) is **connected** if the graph \( (\mathcal{E}, \rightarrow \cup \triangleright) \) is connected. A **connected component** of \( \mathcal{M} \) is a split-CBM induced by a connected component of \( (\mathcal{E}, \rightarrow \cup \triangleright) \).

An **atomic** split-CBM is a connected split-CBM with no process edges \( (\rightarrow = \emptyset) \). Hence, an atomic split-CBM consists of either a single internal event, or a pair of events linked with a matching edge from \( \triangleright \).

We may also describe a split-CBM as a tuple \( \mathcal{M} = ((w^1_p, \ldots, w^\ell_p)p \in \text{Procs}, (\triangleright^d)_{d \in DS}) \) with \( w^j_p \in \Sigma^+ \) such that \( \mathcal{M} = ((w_p)p \in \text{Procs}, (\triangleright^d)_{d \in DS}) \) is a CBM, where \( w_p = w^1_p \cdots w^\ell_p \) for each \( p \in \text{Procs} \). The width of \( \mathcal{M} \) is \( \sum_{p \in \text{Procs}} \ell_p \).

**Example 4.14.** The split-CBM \( \mathcal{M}' \) depicted below has 3 holes and 5 factors. It has two connected components: \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \). It may be obtained with 3 splits either from the CBM \( \mathcal{M} \) or from the CBM \( \mathcal{M}'' \) below.

\[\begin{array}{c}
\mathcal{M}' \\
\mathcal{M}_1 \\
\mathcal{M} \\
\mathcal{M}_2 \\
\mathcal{M}''
\end{array}\]

The **split-game** is a two player turn based game \( \mathcal{G} = (V_\exists \cup V_\forall, E) \) where Eve’s set of positions \( V_\exists \) consists of all connected split-CBMs and Adam’s set of positions \( V_\forall \) consists of non-connected split-CBMs. The edges \( E \) of \( \mathcal{G} \) reflect the moves of the players. Eve’s moves consist in splitting a factor in two, i.e., removing one process.

\(^2\)In [AG14], the width is the number of splits and not the number of factors. For each nonempty process \( p \) \((\mathcal{E}_p \neq \emptyset)\), the number of factors on process \( p \) is one plus the number of splits on process \( p \). Hence, the number of factors is at most the number of splits plus the number of processes.
edge in the graph. Adam’s moves amount to choosing a connected component of the split-CBM. Atomic split-CBMs are terminal positions in the game: neither Eve nor Adam can move from an atomic split-CBM.

A play on a split-CBM $\mathcal{M}$ is path in $\mathcal{G}$ starting from $\mathcal{M}$ and leading to an atomic split-CBM. The cost of the play is the maximum width of any split-CBM encountered in the path. Eve’s objective is to minimize the cost and Adam’s objective is to maximize the cost.

A strategy for Eve from a split-CBM $\mathcal{M}$ can be described with a split-tree $T$ which is a binary tree labeled with split-CBMs satisfying:

1. The root is labeled by $\mathcal{M} = \text{cbm}(T)$.
2. Leaves are labeled by atomic split-CBMs.
3. Eve’s move: Each unary node is labeled with some connected split-CBM $\mathcal{M}$ and its child is labeled with some $\mathcal{M}'$ obtained by splitting a factor of $\mathcal{M}$ in two, i.e., by removing one process edge. Thus, $\text{width}(\mathcal{M}') = 1 + \text{width}(\mathcal{M})$.
4. Adam’s move: Each binary node is labeled with some non connected split-CBM $\mathcal{M} = \mathcal{M}_1 \sqcup \mathcal{M}_2$ where $\mathcal{M}_1$ and $\mathcal{M}_2$ are the labels of its children. Note that $\text{width}(\mathcal{M}) = \text{width}(\mathcal{M}_1) + \text{width}(\mathcal{M}_2)$.

The width of a split-tree $T$, denoted $\text{width}(T)$, is the maximum width of the split-CBMs labeling the tree $T$. A $k$-split-tree is a split-tree of width at most $k$.

**Definition 4.15 (split-width).** The split-width of a split-CBM $\mathcal{M}$ is the minimal width of all split-trees for $\mathcal{M}$.

Let $\text{CBM}^{k\text{-sw}}(\mathfrak{A}, \Sigma)$ denote the set of CBMs with split-width bounded by $k$.

**Example 4.16.** A split-tree is given in Figure 4.1 The intermediate nodes in a sequence of moves by Eve are not shown. The width of the split-tree is 5.

**Remark 4.17.** Any subtree of a split-tree is also a split-tree.

**Example 4.18.** We consider some examples:

- Atomic CBMs have split-width 1 or 2.
- Nested words have split-width $\leq 3$.
- $k$-$\exists B$ CBMs have split-width at most $k + 2$.
- Consider the set of MSCs $L$ suggested in Example 4.2(c). Then, every MSC from $L$ has split-width 4.
Figure 4.1: A split decomposition of width 5.
4.5 Main Results

Consider the following problems:

<table>
<thead>
<tr>
<th>SW-Nonemptiness($\mathcal{A}$, $\Sigma$):</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance:</strong> $S \in \text{CPDS}(\mathcal{A}, \Sigma)$; $k \geq 0$</td>
</tr>
<tr>
<td><strong>Question:</strong> $L(S) \cap \text{CBM}^{k\text{-sw}} \neq \emptyset$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SW-Inclusion($\mathcal{A}$, $\Sigma$):</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance:</strong> $S, S' \in \text{CPDS}(\mathcal{A}, \Sigma)$; $k \geq 0$</td>
</tr>
<tr>
<td><strong>Question:</strong> $L(S) \cap \text{CBM}^{k\text{-sw}} \subseteq L(S')$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SW-Universality($\mathcal{A}$, $\Sigma$):</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance:</strong> $S \in \text{CPDS}(\mathcal{A}, \Sigma)$; $k \geq 0$</td>
</tr>
<tr>
<td><strong>Question:</strong> $\text{CBM}^{k\text{-sw}} \subseteq L(S)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SW-Satisfiability($\mathcal{A}$, $\Sigma$):</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance:</strong> $\varphi \in \text{MSO}(\mathcal{A}, \Sigma)$; $k \geq 0$</td>
</tr>
<tr>
<td><strong>Question:</strong> $L(\varphi) \cap \text{CBM}^{k\text{-sw}} \neq \emptyset$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SW-ModelChecking($\mathcal{A}$, $\Sigma$):</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance:</strong> $S \in \text{CPDS}(\mathcal{A}, \Sigma)$; $\varphi \in \text{MSO}(\mathcal{A}, \Sigma)$; $k \geq 0$</td>
</tr>
<tr>
<td><strong>Question:</strong> $L(S) \cap \text{CBM}^{k\text{-sw}} \subseteq L(\varphi)$</td>
</tr>
</tbody>
</table>

In the following, we will prove the following:

**Theorem 4.19.** All these problems are decidable.

The proof technique is via an interpretation of $\text{CBM}^{k\text{-sw}}$ in binary trees and reduction to problems on tree automata. This is actually similar to decidability for cographs as explained in Section 4.3.

In the following, we introduce

- *special tree terms*: these terms allow to denote graphs such as CBMs. Moreover there is a tree-interpretation of CBMs in special tree terms, viewed as binary trees.

- $\mathcal{A}_{\text{cbm}}^{k\text{-sw}}$: a tree automaton for special tree terms denoting graphs in $\text{CBM}^{k\text{-sw}}$. 

46
4.6 Special tree terms and tree interpretation

In this section, we introduce special tree terms (STTs) and their semantics as labeled graphs. A special tree term using at most \( k \) colors (\( k \)-STT) defines a graph of special tree-width at most \( k - 1 \). Special tree-width is similar to tree-width. See [Cou10] for more details on special tree-width and tree-width. We also give an MSO interpretation of the graph \( G_\tau \) defined by a special tree term \( \tau \) in the binary tree associated with \( \tau \).

A \((\Sigma, \Gamma)\)-labeled graph is a tuple \( G = (V, (E_\gamma)_{\gamma \in \Gamma}, \lambda) \) where \( \lambda : V \rightarrow \Sigma \) is the vertex labeling and \( E_\gamma \subseteq V^2 \) is the set of edges for each label \( \gamma \in \Gamma \).

**Special tree terms** form an algebra to define labeled graphs. The syntax of \( k \)-STTs over \((\Sigma, \Gamma)\) is given by

\[
\tau ::= (i, a) \mid \text{Add}_{i,j}^\gamma \tau \mid \text{Forget}_i \tau \mid \text{Rename}_{i,j} \tau \mid \tau \oplus \tau
\]

where \( a \in \Sigma \), \( \gamma \in \Gamma \) and \( i, j \in \{1, \ldots, k\} \) are colors.

Each \( k \)-STT represents a colored graph \([\tau] = (G_\tau, \chi_\tau)\) where \( G_\tau \) is a \((\Sigma, \Gamma)\)-labeled graph and \( \chi_\tau : [k] \rightarrow V \) is a partial injective function assigning a vertex of \( G_\tau \) to some colors.

- \([(i, a)]\) consists of a single \( a \)-labeled vertex with color \( i \).
- \( \text{Add}_{i,j}^\gamma \) adds a \( \gamma \)-labeled edge to the vertices colored \( i \) and \( j \) (if such vertices exist).
  Formally, if \([\tau] = (V, (E_\gamma)_{\gamma \in \Gamma}, \lambda, \chi)\) then \([\text{Add}_{i,j}^\gamma \tau] = (V, (E_\gamma')_{\gamma \in \Gamma}, \lambda, \chi')\) with \( E_\gamma' = E_\gamma \) if \( \gamma \neq \alpha \) and \( E_\alpha' = \begin{cases} E_\alpha & \text{if } \{i, j\} \not\subseteq \text{dom}(\chi) \\ E_\alpha \cup \{(\chi(i), \chi(j))\} & \text{otherwise.} \end{cases} \)
- \( \text{Forget}_i \) removes color \( i \) from the domain of the color map.
  Formally, if \([\tau] = (V, (E_\gamma)_{\gamma \in \Gamma}, \lambda, \chi)\) then \([\text{Forget}_i \tau] = (V, (E_\gamma')_{\gamma \in \Gamma}, \lambda, \chi')\) with \( \text{dom}(\chi') = \text{dom}(\chi) \setminus \{i\} \) and \( \chi'(j) = \chi(j) \) for all \( j \in \text{dom}(\chi') \).
- \( \text{Rename}_{i,j} \) exchanges the colors \( i \) and \( j \).
  Formally, if \([\tau] = (V, (E_\gamma)_{\gamma \in \Gamma}, \lambda, \chi)\) then \([\text{Rename}_{i,j} \tau] = (V, (E_\gamma')_{\gamma \in \Gamma}, \lambda, \chi')\) with \( \chi'(\ell) = \chi(\ell) \) if \( \ell \in \text{dom}(\chi) \setminus \{i, j\} \), \( \chi'(i) = \chi(j) \) if \( j \in \text{dom}(\chi) \) and \( \chi'(j) = \chi(i) \) if \( i \in \text{dom}(\chi) \).
• Finally, $\oplus$ constructs the disjoint union of the two graphs provided they use different colors. This operation is undefined otherwise.

Formally, if $[\tau_i] = (G_i, \chi_i)$ for $i = 1, 2$ and $\text{dom}(\chi_1) \cap \text{dom}(\chi_2) = \emptyset$ then $[\tau_1 \oplus \tau_2] = (G_1 \uplus G_2, \chi_1 \uplus \chi_2)$. Otherwise, $\tau_1 \oplus \tau_2$ is not a valid STT.

The special tree-width of a graph $G$ is the least $k$ such that $G = G_\tau$ for some $(k + 1)$-STT $\tau$.

For CBMs, we have process edges and data edges, so we take $\Gamma = \{\rightarrow\} \cup \text{DS}$. Also, vertices of CBMs are labeled with a letter from $\Sigma$ and a process from $\text{Procs}$. Hence the labels of vertices are pairs in $\Sigma \times \text{Procs}$ and the atomic STTs used to define CBMs are of the form $(i, a, p)$ with $i \in [k]$, $a \in \Sigma$ and $p \in \text{Procs}$.

**Example 4.20.** The 4-STT below defines the CBM $M_2$ of Figure 4.1.

$$
\tau = \text{Forget}_2 \text{Add}^\rightarrow_{2,4} \text{Add}^\rightarrow_{3,2} \text{Add}^\oplus_{1,2}\big((1, a, q) \oplus (2, c, p)\big) \oplus \text{Add}^\oplus_{3,4}\big((3, b, p) \oplus (4, d, p)\big)
$$

![Tree Diagram](image)

We show now that a graph $G$ defined by a STT $\tau$ can be interpreted in the binary 6th Lecture tree $\tau$. Notice that the vertices of $G$ are in bijection with the leaves of $\tau$. The main difficulty is to interpret the edge relations $E_\gamma$ of $G$ in the tree $\tau$.

Let us denote by $\Lambda^k$ the alphabet of $k$-STTs:

$$
\Lambda^k = \{\oplus, (i, a), \text{Add}^\gamma_{i,j}, \text{Forget}_i, \text{Rename}_{i,j} \ | \ i, j \in [k], a \in \Sigma, \gamma \in \Gamma\}.
$$

Clearly, the set of $k$-STTs considered as binary trees over $\Lambda^k$ is a regular tree language.

**Lemma 4.21.** There is a formula $\Phi_{\text{valid}}^{k-\text{stt}} \in \text{MSO}(\Lambda^k, \downarrow_0, \downarrow_1)$ which defines the set of valid $k$-STTs.

**Proof.** First, the binary tree is correctly labeled: leaves should have labels in $[k] \times \Sigma$, unary nodes should have labels in $\{\text{Add}^\gamma_{i,j}, \text{Forget}_i, \text{Rename}_{i,j} \ | \ i, j \in [k] \text{ and } \gamma \in \Gamma\}$ and binary nodes should be labeled $\oplus$. Moreover, for the $k$-STT to be valid, the
children of a binary node should have disjoint sets of active colors. This can be expressed with

$$\neg \exists x, y, z \bigvee_{1 \leq i, j, \ell \leq k, a, b \in \Sigma} P_{(i,a)}(x) \land P_{(j,b)}(y) \land \text{Same}^k_{i,\ell}(x, z) \land \text{Same}^k_{j,\ell}(y, z) \land \oplus(z)$$

where \(\text{same}^k_{i,\ell}(x, z)\) is a macro stating that \(z\) is an ancestor of \(x\) in the tree and that the vertex colored \(i\) at node \(x\) of the STT is colored \(\ell\) at node \(z\). This formula can be written in MSO with a transitive closure.

**Proposition 4.22 (MSO interpretation).** For all sentences \(\varphi \in \text{MSO}(\Sigma, \Gamma)\) and all \(k > 0\), there is a sentence \(\tilde{\varphi}^k \in \text{MSO}(\Lambda^k, \downarrow_0, \downarrow_1)\) such that, for every \(k\)-STT \(\tau\) with \([\tau] = (G, \chi)\), we have

$$G \models \varphi \iff \tau \models \tilde{\varphi}^k.$$

**Proof.** We proceed by induction on \(\varphi\). Hence we also have to deal with free variables. We denote by \(I\) an interpretation of variables to (sets of) vertices of \(G\) which are identified with leaves of \(\tau\). Hence, we prove by induction that for all formulas \(\varphi \in \text{MSO}(\Sigma, \Gamma)\) and all \(k > 0\), there is a formula \(\tilde{\varphi}^k \in \text{MSO}(\Lambda^k, \downarrow_0, \downarrow_1)\) such that, for all interpretations \(I\) in vertices of \(G\), we have

$$G \models_{\downarrow} \varphi \iff \tau \models_{\downarrow} \tilde{\varphi}^k.$$

The difficult case is to translate the edge relations. We define

$$\bar{x} E_\gamma y = \exists z \bigvee_{1 \leq i, j, \ell \leq k, 1 \leq i', j', \ell' \leq k, a, b, a', b' \in \Sigma} P_{(i,a)}(x) \land P_{(j,b)}(y) \land \text{Same}^k_{i',\ell'}(x, z) \land \text{Same}^k_{j',\ell'}(y, z) \land \text{Add}^\gamma_{i',j'}(z).$$

The other cases are easy:

$$\bar{P}^k_{a}(x) = \bigvee_{1 \leq i \leq k} P_{(i,a)}(x) \quad \bar{\neg} \varphi = \neg \bar{\varphi} \quad \text{Same}^{\varphi_1 \lor \varphi_2} = \text{Same}^{\varphi_1} \lor \text{Same}^{\varphi_2} \quad \exists x \bar{\varphi} = \exists x (\varphi \land \text{leaf}(x)) \quad \exists X \bar{\varphi} = \exists X (\varphi \land \forall x(x \in X \rightarrow \text{leaf}(x))) \quad \bar{x} = y = (x = y)$$

This concludes the proof.

**Corollary 4.23.** The MSO theory of graphs of special tree-width at most \(k\) is decidable.

**Proof.** Let \(\varphi \in \text{MSO}(\Sigma, \Gamma)\). Then, \(\varphi\) is valid on graphs of special tree width at most \(k-1\) iff \(\varphi^{\text{valid}}_{\text{valid}} \rightarrow \tilde{\varphi}^k\) is valid on binary trees. The result of the corollary follows, since MSO validity is decidable on binary trees. Indeed, the problem can be reduced to tree-automata emptiness [TW68]. Note that, moreover, \(\varphi\) is satisfiable on graphs of special tree width at most \(k-1\) iff \(\varphi^{\text{valid}}_{\text{valid}} \land \tilde{\varphi}^k\) is satisfiable on binary trees. ■
Exercise 4.24. Construct a tree automaton $A^k$ with $O(k^2)$ states which accepts a $k$-STT $\tau$ with two marked leaves $x$ and $y$ iff there is a $\gamma$-edge between $x$ and $y$ in the graph $[\tau]$.

Solution: $A^k$ is a deterministic bottom-up tree automaton. It keeps in its state a pair of colors $(i,j) \in \{0, 1, \ldots, k\}^2$ where $i$ is the color at the current node of leaf $x$, with $i = 0$ if $x$ is not in the current subtree. Same for $j$ and $y$. The state is initialized at leaves. It is updated at $\oplus$-nodes and Rename-nodes. The automaton goes to an accepting state if it is in state $(i,j)$ when reading a node labeled $\text{Add}_{i,j}$. On the other hand, it goes to a rejecting state at a node $\text{Forget}_\ell$ if it is in state $(i,j)$ with $\ell \in \{i, j\}$.

Exercise 4.25. Construct a walking tree automaton $B^k$ with $O(k)$ states which runs on a $k$-STT $\tau$ starting from a leaf (say $x$) and accepts when reaching a leaf (say $y$) such that there is a $\gamma$-edge between $x$ and $y$ in the graph $[\tau]$.

Solution: First, walking up the tree, the automaton $B^k$ keeps in its state the color of the leaf $x$. It updates the color at Rename-nodes, makes sure that the color is not forgotten, until it reaches a node labeled $\text{Add}_{i,j}$ where $i$ is the color in its state. Then, it updates its state with color $j$ and it enters a second phase where it walks down the tree (non-deterministically at $\oplus$-nodes), updating the color at Rename-nodes, making sure that the color is not forgotten, until it reaches a leaf having the color which is in its state.

Remark: The automaton $B^k$ can be made deterministic if there is at most one $\gamma$ edge with source $x$. Indeed, walking up the tree is deterministic and we can search for the target leaf $y$ using a DFS.

4.7 Split-width and Special tree-width

Here we show that CBMs of bounded split-width are graphs of bounded special tree-width. Actually, the converse also holds: CBMs of bounded special tree-width have bounded split-width (see [Cyr14]), but we do not need this direction for the decision procedures.

Let $\mathcal{M} = (E, \rightarrow, (\triangleright d)_{d \in DS}, \text{pid}, \lambda)$ be a split-CBM. We denote by $\text{EP}(\mathcal{M})$ the subset of events that are endpoints of factors of $\mathcal{M}$. A left endpoint is an event $e \in E$ such that there are no $f$ with $f \rightarrow e$. We define similarly right endpoints. Note that an event may be both a left and right endpoint. The number of endpoints is at most twice the number of factors: $|\text{EP}(\mathcal{M})| \leq \text{width}(\mathcal{M})$.

We associate with every split-tree $T$ of width at most $k$ a $2k$-STT $\mathcal{T}$ such that $[\mathcal{T}] = (\mathcal{M}, \chi)$ where $\mathcal{M} = \text{cbm}(\mathcal{T})$ is the label of the root of $T$ and the range of $\chi$ is the set of endpoints of $\mathcal{M}$: $\text{Im}(\chi) = \text{EP}(\mathcal{M})$. Notice that $\text{dom}(\chi) \subseteq [2k]$ since $\mathcal{T}$ is a $2k$-STT. The construction is by induction on $T$.

Assume that $\text{cbm}(\mathcal{T})$ is atomic. Then it is either an internal event labeled $(a, p) \in \Sigma \times \text{Procs}$. Then we let $\mathcal{T} = (1, a, p)$. Or, it is a matching write/read pair of events...
If the root of $T$ is a binary node and the left and right subtrees are $T_1$ and $T_2$ then $\text{cbm}(T) = \text{cbm}(T_1) \sqcup \text{cbm}(T_2)$. By induction, for $i = 1, 2$ the STT $T_i$ is already defined and we have $[T_i] = (\text{cbm}(T_i), \chi_i)$. We first rename colors that are active in both STTs. To this end, we choose an injective map $f: \text{dom}(\chi_1) \cap \text{dom}(\chi_2) \rightarrow [2^k] \setminus (\text{dom}(\chi_1) \cup \text{dom}(\chi_2))$. This is possible since $|\text{dom}(\chi_i)| = |\text{Im}(\chi_i)| = |\text{EP}(\text{cbm}(T_i))|$. Hence, $|\text{dom}(\chi_1)| + |\text{dom}(\chi_2)| = |\text{EP}(\text{cbm}(T))| \leq 2k$.

Assuming that $\text{dom}(f) = \{i_1, \ldots, i_m\}$, we define

$$
\overline{T} = T_1 \oplus \text{Rename}_{i_1,f(i_1)} \cdot \cdots \cdot \text{Rename}_{i_m,f(i_m)} T_2.
$$

Finally, assume that the root of $T$ is a unary node with subtree $T'$. Then, $\text{cbm}(T')$ is obtained from $\text{cbm}(T)$ by splitting one factor, i.e., removing one process edge, say $e \rightarrow f$. We deduce that $e$ and $f$ are endpoints of $\text{cbm}(T')$, respectively right and left endpoints. By induction, the STT $T'$ is already defined. We have $[T'] = (\text{cbm}(T'), \chi')$ and $e, f \in \text{Im}(\chi')$. So let $i, j$ be such that $\chi'(i) = e$ and $\chi'(j) = f$. We add the process edge with $\text{Add}_{i,j} T'$. Then we forget color $i$ if $e$ is no more an endpoint, and we forget $j$ if $f$ is no more an endpoint.

**Example 4.26.** Consider the split-tree rooted at node $M_2$ in Figure 4.1. The associated 4-STT is given below.

![Diagram](image)

**Corollary 4.27.** A CBM of split-width $k$ has special tree-width at most $2k$. 

51
Recall that for an STT $\tau$ we write $[\tau] = (G_{\tau}, \chi_{\tau})$.

We have seen in the previous section that all CBMs of split-width at most $k$ can be denoted with $2k$-STTs. On the other hand, not all $2k$-STTs $\tau$ define graphs $G_{\tau}$ which are CBMs, and even if $G_{\tau}$ is a CBM we do not know its split-width. In order to use the tree interpretation in STTs to solve problems on CBMs of bounded split-width, the following result is crucial.

**Proposition 4.28.** There is a tree automaton $A_{cbm}^{k\text{-sw}}$ of size $2^{O(k^2|\mathcal{A}|)}$ such that $L(A_{cbm}^{k\text{-sw}}) \subseteq 2k$-STT and $\text{CBM}^{k\text{-sw}} = \{ G_{\tau} \mid \tau \in L(A_{cbm}^{k\text{-sw}}) \}$.

Notice that, to satisfy the statement above, $A_{cbm}^{k\text{-sw}}$ needs not accept all $2k$-STTs $\tau$ such that $G_{\tau}$ is a CBM of split-width at most $k$ (and the automaton constructed in the proof below will not). But, for each $M \in \text{CBM}^{k\text{-sw}}$, the automaton $A_{cbm}^{k\text{-sw}}$ should accept at least one $2k$-STT $\tau$ such that $G_{\tau} = M$. To this end, we will construct $A_{cbm}^{k\text{-sw}}$ so that it accepts all $2k$-STTs $T$ associated with $k$-split-trees $T$ (See Section 4.7).

Notice also that $A_{cbm}^{k\text{-sw}}$ checks two independent properties, being a CBM and having split-width at most $k$. The first one can be expressed by an MSO$(\exists, \Sigma)$ formula. Hence, using the MSO-interpretation of Proposition 4.22, we can obtain an equivalent tree automaton on $2k$-STTs. But to achieve the size of the automaton stated in the proposition above, we give a (non-trivial) direct construction.

**Proof.** The tree automaton $A_{cbm}^{k\text{-sw}}$ will accept $2k$-STTs of the following form

$$\tau ::= \text{atomicSTT} \mid \text{Add}_{i,j} \tau \mid \text{Forget}_i \tau \mid \text{Rename}_{i,j} \tau \mid \tau \oplus \tau$$

in which write/read matching pairs are given by atomic STTs and later only process edges are added. Hence, the bottom-up automaton $A_{cbm}^{k\text{-sw}}$ first checks that the tree is build from atomic $2k$-STTs. Then, it keeps in its state a tuple $s = (F, \text{pid}, c^-, c^+, (R_d)_{d \in \mathcal{D}}, B)$ where:

- $F$ is the set of factors in the graph (split-CBM) associated with the current subterm. Since we want the split-width to be bounded by $k$, we will have $|F| \leq k$ so we may assume $F \subseteq [k]$.
  
  We denote by $\text{left}(x)$ and $\text{right}(x)$ the left and right endpoints of factor $x$ in the associated graph.

- $\text{pid}: F \rightarrow \text{Proc}$ the process of each factor.

- $c^-, c^+: F \rightarrow [2k]$ the colors of the left/right endpoints of the factors.

- $\forall d \in \mathcal{D}$, a relation $R_d \subseteq [n]^2$.
  
  We will maintain as an invariant the fact that a pair $(x, y)$ of factors belongs to $R_d$ if $x \neq y$ and in the associated graph there is an edge $e \triangleright^d f$ from some event $e$ in factor $x$ to some event $f$ in factor $y$. 

52
• $B \subseteq F^2 \subseteq [k]^2$ an irreflexive binary relation on factors: $x B y$ means that factor $x$ is before factor $y$. This relation will be used to check that the LIFO and FIFO conditions are respected and that there are no cycles in the relation $\leq (\rightarrow \cup \triangleright)^+$ of the split-CBM. The invariant is that for all $x, y \in F$, if $x \neq y$ and $\text{left}(x) < \text{right}(y)$ in the associated graph then $x B y$. The before relation should satisfy the following:

1. If $(x, y) \in R_d$ then $x B y$.
2. $B$ is a total order on the factors $\pi^{-1}(p) \subseteq F$ of each process $p \in \text{Procs}$.
3. $B$ needs not be an order relation. It should be irreflexive but it needs not be antisymmetric (the first condition above may impose simultaneously $x B y$ and $y B x$). Also, $B$ needs not be transitive but it should satisfy the following closure properties: $\forall x, y, y', z \in F$ with $\pi(y) = \pi(y')$, we have
   \[
   x B y B y' B z \land x \neq z \Rightarrow x B z
   \]
   \[
   x B y B y' \land x \neq y' \Rightarrow x B y'
   \]
   \[
   y B y' B z \land y \neq z \Rightarrow y B z
   \]

Finally, $B$ should respect the FIFO and LIFO conditions:

4. FIFO: $\forall d \in \text{Queues}$, for all $(x, y), (x', y') \in R_d$ if $x B x'$ then $\neg(y' B y)$.
5. LIFO: $\forall d \in \text{Queues}$, for all $(x, y), (x', y') \in R_d$, $\neg(x B x' B y B y')$.

The number of states of $A^{k,sw}_{\text{cbm}}$ is thus $2^{O(k^2|\mathcal{A}|)}$.

Now, we describe the transitions of $A^{k,sw}_{\text{cbm}}$.

• the state at an atomic STT of the form $\text{Add}_{i,j}^d(-\oplus-)$ can easily be computed. In particular, we have $F = \{1, 2\}$, $\pi(1) = \text{Writer}(d)$, $\pi(2) = \text{Reader}(d)$, $c^-(1) = c^+(1) = 1$, $c^-(2) = c^+(2) = 2$, $R_d = \{(1, 2)\}$ and $B = \{(1, 2)\}$.

• If the current node is labeled $\text{Add}_{i,j}^-$. Let $s = (F, \pi, c^-, c^+, (R_d)_{d \in \text{DS}}, B)$ be the state at the child. We check that there are two factors $x, y \in F$ such that $c^+(x) = i, c^-(y) = j$ and $x, y$ are consecutive factors on process $p = \pi(x) = \pi(y)$ according to the guessed ordering $B$: we have $x B y$ and $\neg(x B z B y)$ for all $z \in F$ with $\pi(z) = p$.

The new state $s' = (F', \pi', c'^-, c'^+, (R'_d)_{d \in \text{DS}}, B')$ is computed as follows. We use $x$ to represent the merged factor, so we let $F' = F \setminus \{y\}$. $\pi$'s are preserved. Then, $c'^+(x) = c^+(y)$ and the other colors are unchanged. For each $d \in \text{DS}$ we let $R'_d = R_d \cup \{(z, x) \mid z \neq x \land (z, y) \in R_d\} \cup \{(x, z) \mid (y, z) \in R_d\}$. Finally, $B' = (B \cap F'^2) \cup \{(z, x) \mid x \neq z B y\}$.

We can check that the new state $s'$ satisfies all conditions (1-5) on $B'$.
Assume the current node is labeled $\oplus$ hence the subterm is $\tau = \tau_1 \oplus \tau_2$. Let $s_1$ and $s_2$ be the states at $\tau_1$ and $\tau_2$ respectively. We may rename the factors in $s_2$ so that $F_1 \cap F_2 = \emptyset$. The state $s$ is obtained from $s_1$ and $s_2$ by taking the union of its components: $F = F_1 \cup F_2$, $\text{pid} = \text{pid}_1 \cup \text{pid}_2$, etc. For the before relation, we guess $B \subseteq F^2$ which satisfies all conditions (1-5) and which is compatible with the before relations of $s_1$ and $s_2$, i.e., $B_1 \cup B_2 \subseteq B$. In particular, we have to guess for each process $p \in \text{Procs}$ how factors of $\tau_1$ and $\tau_2$ on process $p$ will be shuffled.

- If the current node is labeled $\text{Forget}$, and the state at its child is $s = (F, \text{pid}, c^-, c^+, (R_d)_{d \in \text{DS}}, B)$ then we check that $i \notin c^-(F) \cup c^+(F)$: the color of an endpoint should not be forgotten. The new state is simply $s$.

- If the current node is labeled $\text{Rename}_{i,j}$ and the state at its child is $s = (F, \text{pid}, c^-, c^+, (R_d)_{d \in \text{DS}}, B)$ then we exchange $i$ and $j$ in $c^-$ and $c^+$ to obtain the new state.

A state $s = (F, \text{pid}, c^-, c^+, (R_d)_{d \in \text{DS}}, B)$ is accepting if we have at most one factor per process.

Claim 4.29. If a 2k-STT $\tau$ admits an accepting run in $A_{\text{cbm}}^{k-\text{sw}}$ then the graph $G_\tau$ is a CBM and its split-width is at most $k$.

Assume that $A_{\text{cbm}}^{k-\text{sw}}$ has an accepting run on $\tau$ and let $G_\tau = (\mathcal{E}, \rightarrow, (\triangleright^d)_{d \in \text{DS}}, \text{pid}, \lambda)$ be the graph defined by $\tau$. We have to check that $G_\tau$ satisfies all conditions of Definition 2.9. First, it is clear from atomic STTs that $\triangleright^d \subseteq \mathcal{E}_{\text{Writer}(d)} \times \mathcal{E}_{\text{Reader}(d)}$ and that matching edges are pairwise disjoint. Also, we add $\rightarrow$ edges only between events of the same process.

We can check by induction that, at each subterm $\tau$ which is labeled with state $s = (F, \text{pid}, c^-, c^+, (R_d)_{d \in \text{DS}}, B)$ in the accepting run, we have for all $x, y \in F$, if $x \neq y$ and $\text{left}(x) < \text{right}(y)$ in the associated graph $G_\tau$ then $x \leq B \geq y$. Since the disjoint union induced by $\oplus$ does not add any ordering between the split-CBMs defined by the two children, the above condition is satisfied by induction as soon as $B_1 \cup B_2 \subseteq B$. We can also show by induction that the property is preserved at $\text{Add}_{i,j}$ nodes using condition (3) of the before relations.

Next, we show that $<$ in $G_\tau$ is antisymmetric. Towards a contradiction assume that at some $\text{Add}_{i,j}$ node, we add a process edge $\text{right}(x) \rightarrow \text{left}(y)$ and that we had $\text{left}(y) < \text{right}(x)$ at the child node. By the property above of $B$ this implies $y \leq B \geq x$ at the child. This is impossible since the transition checks that $x \leq B \geq y$ and, by condition 2, $B$ is a total order on the factors of every process.

It is easy to check by induction that $\rightarrow$ defines a total order on the events of each factor. Since at the root we are in an accepting state, there are at most one factor per process. Hence, $\rightarrow$ defines a total order on $\mathcal{E}_p$ for all processes $p \in \text{Procs}$.

Finally, we can check by induction that for all data-structures $d \in \text{DS}$, the relation $R_d$ in a state consists in the set of pairs of factors $x \neq y$ such that $e \triangleright^d f$ for some events $e$ in factor $x$ and $f$ in factor $y$ of the associated graph.
Let $d \in \text{Queues}$ and $e_1 \triangleright^d f_1, e_2 \triangleright^d f_2$ in $G_\tau$ with $e_1 < e_2$. The least common ancester of the atomic STTs corresponding to these matching pairs is a node labeled $\oplus$. At that node, the four factors $x_1, y_1, x_2, y_2$ of the events $e_1, f_1, e_2, f_2$ are distinct. Since $e_1 < e_2$ we get $x_1 B x_2$ by the above property. If we had $f_2 < f_1$ then we would get $y_2 B y_1$, again by the above property. But this is not allowed by the transition at a $\oplus$ node since the guessed relation $B$ satisfies condition (4).

The proof is similar for stacks. Let $d \in \text{Stacks}$ and $e_1 \triangleright^d f_1, e_2 \triangleright^d f_2$ in $G_\tau$. Assume that $e_1 < e_2 < f_1 < f_2$. The the least common ancestor is again a $\oplus$ node and from the ordering above we deduce that the corresponding factors should satisfy $x_1 B x_2 B y_1 B y_2$, a contradiction.

We deduce that $G_\tau$ is a CBM. From the 2k-STT $\tau$ we obtain a strategy for Eve in the split-game, i.e., a split-tree $T$ for $G_\tau$. We need not have $\tau = \overline{T}$, but the labels of the nodes of $T$ correspond to graphs defined by subterms of $\tau$. Since these graphs have at most $k$ factors, we deduce that the width of $T$ is at most $k$. Hence, $G_\tau \in \text{CBM}^{k-\text{sw}}$ which concludes the proof of the claim.

We deduce that $\{G_\tau \mid \tau \in L(A_{\text{cbm}}^{k-\text{sw}})\} \subseteq \text{CBM}^{k-\text{sw}}$. The converse inclusion follows from the next claim.

**Claim 4.30.** If $T$ is a $k$-split tree for some CBM $\overline{M}$ then the associated 2k-STT $\overline{T}$ is accepted by $A_{\text{cbm}}^{k-\text{sw}}$.

Each split-CBM $M$ labeling some node of $T$ has an induced before relation on its factors inherited from $\overline{M}$: if $x \neq y$ are factors of $M$ then $x B_M y$ if left($x$) $<$ right($y$) in $\overline{M}$. We can easily check that $B_M$ satisfies conditions (1–5).

The run of $A_{\text{cbm}}^{k-\text{sw}}$ on $T$ is constructed bottom-up. All transitions are deterministic except for $\oplus$ nodes. Consider a subterm $\tau = \tau_1 \oplus \tau_2$ of $\overline{T}$. Then $G_\tau$ is a split-CBM labeling the corresponding node in $T$. We use the relation $B_{G_\tau}$ to construct the transition at this $\oplus$ node.

We can check that we obtain a valid run of $A_{\text{cbm}}^{k-\text{sw}}$ on $\overline{T}$. In particular, we can check by induction that for all subterms $\tau$ of $\overline{T}$ the before relation in the state at $\tau$ is exactly $B_{G_\tau}$. We deduce that, when a process edge is added between two factors, then these factors are always consecutive factors on some process, hence transitions at $\text{Add}_{i,j}$ nodes are valid. Also, the color of an endpoint is never forgotten in $\overline{T}$, hence transitions at $\text{Forget}_t$ nodes are valid.

This concludes the proof of Proposition 4.28.

**Proposition 4.31.** Given $k > 0$ and an MSO formula $\varphi \in \text{MSO}(\mathfrak{A}, \Sigma)$, we can construct a tree automaton $A_{\varphi}^{k-\text{stt}}$ such that for all k-STT $\tau$ we have

$$\tau \in L(A_{\varphi}^{k-\text{stt}}) \quad \text{iff} \quad G_\tau \models \varphi.$$

**Proof.** From $\varphi \in \text{MSO}(\mathfrak{A}, \Sigma)$, we construct $\bar{\varphi}^{2k} \in \text{MSO}(A^{2k}, 4_0, 4_1)$ using Proposition 4.22. Using [TW68] we obtain an equivalent tree automaton $A_{\varphi}^{k-\text{stt}}$. 

**Corollary 4.32.** The problem $\text{SW-SATISFIABILITY}(\mathfrak{A}, \Sigma)$ is decidable.
Proposition 4.28. We obtain \( \tau_{M} \in \text{accepting} \), where \( \tau \in L(\bar{\varphi}) \cap \text{CBM}^{k-\text{sw}} \neq \emptyset \) iff \( L(A^{k-\text{sw}}_{\text{cbm}} \cap A^{k-\text{sw}}_{\bar{\varphi}}) \neq \emptyset \).

Indeed, given \( \mathcal{M} \in \text{CBM}^{k-\text{sw}} \) such that \( \mathcal{M} \models \varphi \), we find \( \tau \in L(A^{k-\text{sw}}_{\text{cbm}}) \subseteq 2k-\text{STT} \) with \( G_{\tau} = \mathcal{M} \) by Proposition 4.28. We obtain \( \tau \in L(A^{2k-\text{stt}}_{\bar{\varphi}}) \) by Proposition 4.31. Conversely, if \( \tau \in L(A^{k-\text{sw}}_{\text{cbm}} \cap A^{2k-\text{stt}}_{\bar{\varphi}}) \) then \( G_{\tau} \in \text{CBM}^{k-\text{sw}} \) by Proposition 4.28 and \( G_{\tau} \in L(\bar{\varphi}) \) by Proposition 4.31.

Proposition 4.33. Given a \( \text{CPDS} \mathcal{S} \), we can construct a tree automaton \( A^{k-\text{sw}}_{\mathcal{S}} \) of size \( |\mathcal{S}|^{O(k^2)} \) such that for all \( \tau \in L(A^{k-\text{sw}}_{\text{cbm}}) \) we have

\[ \tau \in L(A^{k-\text{sw}}_{\mathcal{S}}) \iff G_{\tau} \in L(\mathcal{S}). \]

Proof. Let \( \mathcal{S} = (\text{Locs}, \text{Val}, (\rightarrow_{p})_{p \in \text{Procs}}, \ell_{\text{in}}, \text{Fin}) \) be a \( \text{CPDS} \). The tree automaton \( A^{k-\text{sw}}_{\mathcal{S}} \) is a modified version of \( A^{k-\text{sw}}_{\text{cbm}} \). A state of \( A^{k-\text{sw}}_{\mathcal{S}} \) is a tuple \((F, e^{-}, c^{+}, \rho^{-}, \rho^{+})\) where \( F, e^{-}, c^{+} \) are as in \( A^{k-\text{sw}}_{\text{cbm}} \). The maps \( \rho^{-}, \rho^{+} : F \rightarrow \text{Locs} \) are such that for every factor \( x, \mathcal{S} \) has a run from \( \rho^{-}(x) \) to \( \rho^{+}(x) \) reading factor \( x \).

For an atomic \( \text{STT} (1, a, p) \) for an internal event, we have \( F = \{1\} \) and \( c^{-}(1) = c^{+}(1) = 1 \). We guess a transition \( \ell \xrightarrow{a} \ell' \) and we store the locations \( \rho^{-}(1) = \ell \) and \( \rho^{+}(1) = \ell' \).

For an atomic \( \text{STT} \text{Add}_{1,2}(1, a, p) \oplus (2, b, q) \) for a matching write/read pair, we have \( F = \{1, 2\} \), \( c^{-}(1) = c^{+}(1) = 1 \) and \( c^{-}(2) = c^{+}(2) = 2 \). We guess a value \( v \in \text{Val} \) and two matching transitions \( \ell_{1} \xrightarrow{a, d, v} \ell'_{1} \) and \( \ell_{2} \xrightarrow{b, d, v} q \). We store the locations \( \rho^{-}(1) = \ell_{1}, \rho^{+}(1) = \ell'_{1}, \rho^{-}(2) = \ell_{2} \) and \( \rho^{+}(2) = \ell'_{2} \).

The components \( F, e^{-}, c^{+} \) are updated bottom-up as in \( A^{k-\text{sw}}_{\text{cbm}} \). The locations \( \rho^{-}, \rho^{+} \) are unchanged at \( \text{Forget}_{i} \) and \( \text{Rename}_{i,j} \) nodes. We take the disjoin union of these maps at \( \oplus \) nodes. Finally, consider an \( \text{Add}_{i,j} \) node. Let \((F, c^{-}, c^{+}, \rho^{-}, \rho^{+})\) be the state at the child. We find \( x, y \in F \) such that \( c^{+}(x) = i \) and \( c^{-}(y) = j \). We check that \( \rho^{+}(x) = \rho^{-}(y) \) to ensure that the two runs on factors \( x \) and \( y \) can be concatenated. In the new state, we have \( \rho^{-}(x) = \rho^{-}(x) \) and \( \rho^{+}(x) = \rho^{+}(y) \).

Finally, the automaton \( A^{k-\text{sw}}_{\mathcal{S}} \) accepts if the state at the root has at most one factor per process, and \( \rho^{-}(x) = \ell_{\text{in}} \) for all \( x \in F \), and the tuple \((\ell_{p})_{p \in \text{Procs}} \in \text{Fin} \) is accepting, where \( \ell_{p} = \begin{cases} \rho^{+}(x) & \text{if } x \in F \text{ and } \text{pid}(x) = p \\ \ell_{\text{in}} & \text{if } p \notin \text{pid}(F) \end{cases} \).

Corollary 4.34. The problem \( \text{sw-NONEMPTINESS}(\mathcal{A}, \Sigma) \) is decidable in \( \text{ExpTime} \). The procedure is only polynomial in the size of the \( \text{CPDS} \).

Proof. The problem reduces to checking nonemptiness of a tree automaton. Given \( k > 0 \) and a \( \text{CPDS} \mathcal{S} \) over \( \mathcal{A} \), we have \( L(\mathcal{S}) \cap \text{CBM}^{k-\text{sw}} \neq \emptyset \) iff \( L(A^{k-\text{sw}}_{\text{cbm}} \cap A^{k-\text{sw}}_{\mathcal{S}}) \neq \emptyset \).

Indeed, given \( \mathcal{M} \in L(\mathcal{S}) \cap \text{CBM}^{k-\text{sw}} \), we find \( \tau \in L(A^{k-\text{sw}}_{\text{cbm}}) \) with \( G_{\tau} = \mathcal{M} \) by Proposition 4.28. We obtain \( \tau \in L(A^{k-\text{sw}}_{\mathcal{S}}) \) by Proposition 4.33. Conversely, if \( \tau \in L(A^{k-\text{sw}}_{\text{cbm}} \cap A^{k-\text{sw}}_{\mathcal{S}}) \) then \( G_{\tau} \in \text{CBM}^{k-\text{sw}} \) by Proposition 4.28 and \( G_{\tau} \in L(\mathcal{S}) \) by Proposition 4.33.
Corollary 4.35. The problem \textsc{sw-ModelChecking}(\mathfrak{A}, \Sigma) is decidable in \textsc{Exp-Time}. The procedure is only polynomial in the size of the CPDS.

Proof. The problem reduces to checking non-emptiness of a tree automaton. Given \(k > 0\), a CPDS \(S\), and a formula \(\varphi \in \text{MSO}(\mathfrak{A}, \Sigma)\) we have \(L(S) \cap \text{CBM}^{k-\text{sw}} \subseteq L(\varphi)\) iff \(L(A_{\text{cbm}}^{k-\text{sw}} \cap A_{S}^{k-\text{sw}} \cap A_{\varphi}^{2-\text{stt}}) = \emptyset\). \(\blacksquare\)

4.9 ICPDL model checking

In the previous sections, we showed that model checking CPDSs against MSO formulas is decidable when we restrict to behaviors of bounded split-width. However, the transformation of an MSO formula into a tree automaton is inherently non-elementary, and this non-elementary lower bound is in fact inherited by the model-checking problem. We will, therefore, turn to the logic ICPDL and consider the following problem, for a given architecture \(\mathfrak{A}\) (and alphabet \(\Sigma\)):

\begin{center}
\begin{tabular}{ll}
\hline
\textbf{sw-ICPDL-ModelChecking}(\mathfrak{A}, \Sigma): & \\
\textbf{Instance:} & \(S \in \text{CPDS}(\mathfrak{A}, \Sigma)\); \(\Phi \in \text{ICPDL}(\mathfrak{A}, \Sigma)\); \(k \geq 0\) \\
\textbf{Question:} & \(L(S) \cap \text{CBM}^{k-\text{sw}} \subseteq L(\Phi)\) \\
\hline
\end{tabular}
\end{center}

Here, we suppose that \(k\) is given in unary.

Decidability of this problem follows from decidability of MSO and the statement of Exercise 3.10, saying that every ICPDL formula can be (effectively) translated into an MSO sentence. Unfortunately, this does not give us an elementary upper bound. We will rather proceed as follows:

Proposition 4.36. Given \(\Phi \in \text{ICPDL}(\mathfrak{A}, \Sigma)\), and \(k > 0\), we can construct a non-deterministic tree automaton (NTA) \(A_{\Phi}^{k-\text{sw}}\) of doubly exponential size such that, for all \(\tau \in L(A_{\text{cbm}}^{k-\text{sw}})\) (denoting by \(\text{cbm}(\tau)\) the graph \(G_\tau\) defined by \(\tau\)), we have:

\[\tau \in L(A_{\Phi}^{k-\text{sw}}) \text{ iff } \text{cbm}(\tau) \models \Phi\]

This will yield the following result:

\textbf{Theorem 4.37 ([CGNK14])}. \textsc{sw-ICPDL-ModelChecking}(\mathfrak{A}, \Sigma) is solvable in doubly exponential time (when the bound \(k\) on the split-width is encoded in unary).

This theorem follows from Proposition 4.36 since we have

\bullet \(L(S) \cap \text{CBM}^{k-\text{sw}} \subseteq L(\Phi)\) iff \(L(A_{S}^{k-\text{sw}}) \cap L(A_{\text{cbm}}^{k-\text{sw}}) \cap L(A_{\Phi}^{k-\text{sw}}) = \emptyset\), and
• emptiness of NTAs can be checked in polynomial time.

The rest of this section is devoted to the proof of Proposition 4.36. Towards an NTA for a given ICPDL formula \( \Phi \), we first translate local ICPDL formulas \( \sigma \) into an intermediate model, called alternating two-way (tree) automata (A2As). An A2A “walks” in a tree, similarly to a path formula from ICPDL. In addition, it can spawn several copies of an automaton, which all have to accept the input. This spawning is dual to non-deterministic choice, hence the name alternating.

Recall that 2k-STTs are binary trees over the alphabet \( \Lambda^{2k} \) as defined in Section 4.6. In the following, \( \Omega \) is an arbitrary alphabet for binary trees. When our trees are 2k-STTs, then we will take \( \Omega = \Lambda^{2k} \).

**Definition 4.38 (A2A).** An alternating two-way (tree) automaton (A2A) over an alphabet \( \Omega \) is a triple \( \mathcal{A} = (S, \delta, \iota, \text{acc}) \)

- \( S \) nonempty finite set of states
- \( \iota \in S \) initial state
- \( \delta : S \times \Omega \times 2^D \to B^+ (D \times S) \) is the transition function such that \( \delta(s, \alpha, \beta) \in B^+ (\beta \times S) \) for all \( (s, \alpha, \beta) \in S \times \Omega \times 2^D \)
- \( \text{acc} : S \to \mathbb{N} \) is the acceptance condition, assigning to each state a priority ♦

The transition function \( \delta \) needs some explanation. First, the set

\[ D = \{ \uparrow, \downarrow, \uparrow, \downarrow, \text{id} \} \]

is the set of directions available at a node of a binary tree.\(^3\) This is indicated by the type \( \text{type}(u) \subseteq D \) of a node \( u \). We assume that \( \text{type}(u) \) always contains \( \text{id} \). Moreover, when \( u \) is a leaf, then \( \text{type}(u) \cap \{ \uparrow, \downarrow \} = \emptyset \). When a node has only one child, then we assume that the child is the left successor, following \( \uparrow \). Moreover, \( B^+ (D \times S) \) is the set of positive boolean formulas over \( D \times S \), built according to the grammar

\[
B ::= \text{true} \mid \text{false} \mid (d, s) \mid B \lor B \mid B \land B
\]

where \( (d, s) \in D \times S \) is an atomic formula.

An example formula from \( B^+ (D \times S) \) is \( B = (\uparrow, s_1) \lor ((\downarrow, s_2) \land (\text{id}, s_3)) \). Intuitively, this means that, being at a left child, the A2A can (i) go to the father and continue in state \( s_1 \), or (ii) spawn two copies, one going to the left child and starting in \( s_2 \), the other one staying at the current node and starting in \( s_3 \). We will consider that \( \{(\uparrow, s_1)\} \) and \( \{(\downarrow, s_2), (\text{id}, s_3)\} \) are models of \( B \), in the obvious sense. That is, a model is a set \( M \) of atomic formulas such that \( B \) is evaluated to

\(^3\)For convenience, we write \( \uparrow \) and \( \downarrow \) instead of \( \downarrow_0 \) and, respectively, \( \downarrow_1 \).
true when atoms from $M$ are evaluated to true, and all other atoms are evaluated to false. We then write $M \models B$.

Let $t$ be an $\Omega$-labeled binary tree (say, with set of nodes $V$ and labeling function $\lambda : V \to \Omega$), and let $u \in V$ be a node of $t$. A run of the A2A $A$ on $(t, u)$ is a nonempty finite (unranked) $(V \times S)$-labeled tree $\rho$ such that the following hold:

- the root of $\rho$ is labeled with $(u, i)$
- for all nodes $x$ of $\rho$, say with label $(v, s)$, there is a model
  $$\{(d_1, s_1), \ldots, (d_n, s_n)\} \models \delta(s, \lambda(v), \text{type}(v))$$
  such that $x$ has $n$ children, with labels $(d_1(v), s_1), \ldots, (d_n(v), s_n)$, respectively. Hereby, $d_i(v)$ is defined in the expected manner. For example, $\check{\text{v}}$ is the left child of $v$, whereas $\text{id}(v)$ is $v$ itself.

Since a run is finite, this definition implies that, if a leaf is labeled with some $(v, s)$, then $\delta(s, \lambda(v), \text{type}(v))$ is equivalent to true.

The acceptance condition acc applies to infinite branches: run $\rho$ is accepting if, for every of its infinite branches (starting at the root), say with associated infinite sequence $(u_0, s_0)(u_1, s_1)(u_2, s_2)\ldots$ of labels, the minimal number that occurs infinitely often in the sequence $\text{acc}(s_0)\text{acc}(s_1)\text{acc}(s_2)\ldots$ is even.

Finally, we let $L(A) = \{(t, u) \mid \text{there is an accepting run of } A \text{ on } (t, u)\}$.

We will now translate a node formula $\sigma$ into an A2A $A_{\sigma}$ over $\Lambda^{2k}$ such that, for all $\tau \in L(A^{k,\text{sw}}_{\text{cbm}})$ and all leaves $u$ of $\tau$, the following holds:

$$(\tau, u) \in L(A_{\sigma}) \iff \text{cbm}(\tau), u \models \sigma$$

Recall that in the tree interpretation of $\text{cbm}(\tau)$ in the tree $\tau$, vertices of $\text{cbm}(\tau)$ are identified with leaves of $\tau$. Moreover, $\text{cbm}(\tau), u \models \sigma$ stands for $u \in [\sigma]_{\text{cbm}(\tau)}$.

A2A $A_a$ for $a \in \Sigma$:

We let $A_a = (\{s\}, \delta, s, s \mapsto 1)$ where

$$\delta(s, \alpha, \beta) = \begin{cases} 
true & \text{if } \alpha \in \{(i, a, p) \mid i \in [2k], p \in \text{Procs}\} \\
false & \text{otherwise.}
\end{cases}$$

A2A $A_p$ for $p \in \text{Procs}$:
We let $A_p = (\{s\}, \delta, s, s \mapsto 1)$ where
\[
\delta(s, \alpha, \beta) = \begin{cases} 
\text{true} & \text{if } \alpha \in \{(i, a, p) \mid i \in [2k], a \in \Sigma\} \\
\text{false} & \text{otherwise.}
\end{cases}
\]

**A2A $A_{\sigma_1 \lor \sigma_2}$:**
Suppose $A_{\sigma_i} = (S_i, \delta_i, \iota_i, \text{acc}_i)$, for $i \in \{1, 2\}$.
We let $A_{\sigma_1 \lor \sigma_2} = (S_1 \uplus S_2 \uplus \{\iota\}, \delta, \iota, \text{acc}_1 \cup \text{acc}_2 \cup \{\iota \mapsto 1\})$ where
\[
\delta(s, \alpha, \beta) = \begin{cases} 
(id, \iota_1) \lor (id, \iota_2) & \text{if } s = \iota \\
\delta_1(s, \alpha, \beta) & \text{if } s \in S_1 \\
\delta_2(s, \alpha, \beta) & \text{if } s \in S_2
\end{cases}
\]

**A2A $A_{\neg \sigma}$:**
Suppose $A_{\sigma} = (S, \delta, \iota, \text{acc})$.
We let $A_{\neg \sigma} = (S, \delta', \iota, \text{acc}')$ where
\[
\delta'(s, \alpha, \beta) = \overline{\delta(s, \alpha, \beta)}
\]

Here, the dual $\overline{B}$ of a positive boolean formula is defined as follows:

- $\overline{\text{true}} = \text{false}$
- $\overline{\text{false}} = \text{true}$
- $\overline{(d, s)} = (d, s)$ for all $(d, s) \in D \times S$
- $\overline{B_1 \lor B_2} = \overline{B_1} \land \overline{B_2}$
- $\overline{B_1 \land B_2} = \overline{B_1} \lor \overline{B_2}$

Moreover, we let $\text{acc}'(s) = \text{acc}(s) + 1$ for all $s \in S$.

**A2A $A_{(\pi)\sigma}$:**
This is the most difficult case. Similarly to the translation of a PDL formula into a CPDS (cf. Section 3.4), where a path formula is first translated into a finite automaton, we will first translate $\pi$ into a variant of an A2A, which we call *walking (tree) automaton* (WA). A WA also walks in a tree, but does not have alternation capabilities. The main difference with A2A is that a WA does not recognize pairs $(t, u)$, but rather triples $(t, u, v)$. This reflects the fact that path formulas define binary relations on events.

**Definition 4.39 (WA).** A walking (tree) automaton (WA) over $\Omega$ is a tuple $B = (S, \rightarrow, \iota, f)$
- $S$ nonempty finite set of states
- $\iota \in S$ initial state
- $f \in S$ final state
- $\rightarrow$ is the finite transition relation, containing transitions of two types:
  - $s \xrightarrow{\text{dir}} s'$ where $s, s' \in S$ and $\text{dir} \in D$
  - $s \xrightarrow{A} s'$ where $s, s' \in S$ and $A$ is an A2A over $\Omega$

Taking a transition of type $s \xrightarrow{A} s'$, the WA $B$ checks if, in the given tree, the current node is accepted by $A$. It then moves to state $s'$ but does not change the current node. Otherwise, the semantics of $B$ is defined as expected, and we only describe it intuitively as $L(B) = \{(t, u, v) | B \text{ allows for a “walk” in } t \text{ from } u \text{ to } v, \text{ starting in state } \iota \text{ and ending in state } f}\}$.  

**Exercise 4.40.** Provide a formal definition of $L(B)$.

Our aim is to translate a path formula $\pi$ into a WA $B_\pi$ such that, for all $\tau \in L(\mathcal{A}_{\text{sw}})$ and all leaves $u, v$ of $\tau$, the following holds:

$$(\tau, u, v) \in L(B_\pi) \text{ iff } \text{cbm}(\tau), u, u \models \pi$$

Let us, for the moment, assume that we are already given $B_\pi = (S, \rightarrow, \iota, f)$ and that $B_\pi$ uses, in its transition, the A2As $A_i = (S_i, \delta_i, \iota_i, \text{acc}_i)$ with $i = 1, \ldots, n$. Moreover, suppose that, by induction, we have $A_\sigma = (S_\sigma, \delta_\sigma, \iota_\sigma, \text{acc}_\sigma)$ for a node formula $\sigma$. We define the A2A $A_{(\pi)\sigma} = (S', \delta, \iota, \text{acc})$ as follows:

- $S' = S \uplus S_1 \uplus \ldots \uplus S_n \uplus S_\sigma$
- $\delta(s, \alpha, \beta) = \begin{cases} 
  \delta_i(s, \alpha, \beta) & \text{if } s \in S_i \\
  \delta_\sigma(s, \alpha, \beta) & \text{if } s \in S_\sigma \\
  \bigvee_{s \xrightarrow{\text{dir}} s'} (\text{dir}, s') & \text{if } s \in S \\
  \bigvee_{s \xrightarrow{A} s'} (\text{id}, \iota_i) \land (\text{id}, s') & \text{if } s \in S \\
  \bigvee_{s = f} (\text{id}, \iota_\sigma) & \text{if } s \in S \\
\end{cases}$
- $\text{acc} = \text{acc}_1 \cup \ldots \cup \text{acc}_n \cup \text{acc}_\sigma \cup \{s \mapsto 1 \mid s \in S\}$

It remains to construct the WA $B_\pi$ for an arbitrary path formula $\pi$. We proceed by induction:
WA $B_{\oplus d}$ for $d \in DS$:

$$
\begin{array}{c}
\delta \\
\rightarrow \\
\rightarrow \\
\rightarrow \\
\rightarrow \\
\rightarrow \\
\rightarrow \\
\end{array}
A_{\oplus} \rightarrow A_{\oplus d} \rightarrow A_d \rightarrow f
$$

Here, the A2A $A_{\oplus}$ just checks if the current label is $\oplus$, and $A_{d}$ checks whether the current label is $\text{Add}_{1,2}^d$ (similarly to $A_a$ and $A_p$ for the atomic node formulas $a$ and $p$).

WA $B_{\rightarrow}$: We use Exercise 4.25.

WA $B_{\text{test}(\sigma)}$:

$$
\begin{array}{c}
\delta \\
\rightarrow \\
\rightarrow \\
\rightarrow \\
\rightarrow \\
\rightarrow \\
\rightarrow \\
\end{array}
A_{\sigma} \rightarrow f
$$

Here, $A_{\sigma}$ is given by induction.

Converse and regular-expression operators:

The construction of WAs $B_{\pi^{-1}}, B_{\pi_1+\pi_2}, B_{\pi_1\cdot\pi_2},$ and $B_{\pi^*}$ follows the standard methods for finite automata.

WA $B_{\pi_1 \cap \pi_2}$:

This is the most difficult case. The idea for the construction is due to Göller, Lohrey, and Lutz [GLL09].

One might be tempted to apply some product construction to the automata $B_{\pi_1}$ and $B_{\pi_2}$ so that they evolve synchronously. However, recall that, in a tree $\tau \in L(A_{\text{cbm}}^{b\text{-sw}})$, $B_{\pi_1 \cap \pi_2}$ should identify the pairs of leaves $u$ and $v$ such that $\text{cbm}(\tau), u, v \models \pi_i$ for all $i \in \{1, 2\}$. But, the paths leading from $u$ to $v$ that match $\pi_1$ and $\pi_2$, respectively, may be quite different. In particular, there may be no paths matching $\pi_1$ and matching also $\pi_2$, or the other way around, so that a product construction would be too restrictive. This is illustrated in the following figure, where we assume that the red path matches $\pi_1$ and the green one matches $\pi_2$:
The crucial observation is now that both paths have to visit the unique shortest path between \(u\) and \(v\), which is illustrated by the thick blue line. Moreover, when one path leaves this shortest path, then it has to come back along the same way so that, actually, any deviation from the shortest path forms a loop. In the figure, the paths are abstracted as additional cycles.

This motivates the following construction of \(B_{\pi_1 \cap \pi_2}\): Essentially, we apply a product construction to \(B_{\pi_1}\) and \(B_{\pi_2}\) (since there is always the common shortest path). This makes sure that the two paths that we simulate lead to the same node \(v\). However, at any time, any of the \(B_{\pi_i}\) is allowed to perform a loop. To make sure that we simulate a given WA \(B_{\pi_i}\) and the detour ends in the departure node, we make use of the power of alternation in A2As.

**Loops:** So, suppose \(B = (S, \to, \iota, f)\) is any WA (later it will be \(B_{\pi_1}\) or \(B_{\pi_2}\), though with variable initial and final states). Assume \(A_i = (S_i, \delta_i, \iota_i, acc_i)\) with \(i = 1, \ldots, n\) are the A2As that occur in \(B\). We will construct an A2A \(A_B\) such that the following holds, for all \(\Omega\)-labeled trees \(t\) and nodes \(u, v\) of \(t\):

\[
(t, u) \in L(A_B) \quad \text{iff} \quad (t, u, u) \in L(B)
\]

We let \(A_B = (S', \delta, \iota', acc)\) where

- \(S' = S_1 \uplus \ldots \uplus S_n \uplus (S \times S)\)
- \(\iota' = (\iota, f)\)
- \(\delta(s, \alpha, \beta) = \delta_i(s, \alpha, \beta) \quad \text{if} \quad s \in S_i\)
- \(acc = acc_1 \cup \ldots \cup acc_n \cup \{(s, s') \mapsto 1 \mid (s, s') \in S \times S\}\)

The idea is now that a current state \((s, r) \in S \times S\) will guarantee that, in \(B\), we can make a loop while going from state \(s\) to state \(s'\). Accordingly, the remaining transitions are defined as follows. First, \(\delta((s, s), \alpha, \beta) = \text{true}\). Next, if \(s \neq r\) then

\[
\delta((s, r), \alpha, \beta) = \left( \begin{array}{c}
\lor \bigvee_{s \to s'} (\text{id}, \iota_i) \land (\text{id}, (s', r)) \\
\lor \bigvee_{s \xrightarrow{\text{dir}}} s' (\text{dir}, (s', r')) \\
\lor \bigvee_{q \in S} (\text{id}, (s, q)) \land (\text{id}, (q, r)) \\
\lor \bigvee_{s=r} \text{true}
\end{array} \right)
\]

The third case allows a transitive closure of loops. The interesting case is the second one. Whenever \(B\) leaves a node \(u\) via direction \(\text{dir}\) towards a node \(v\), it has to come back to \(u\) along the same edge, passing via \(v\) and taking the dual direction \(\overline{\text{dir}}\). This is actually why the characterization of loops in terms of \(\delta\) is complete. Here, the dual direction is defined by \(\overline{\nearrow} = \searrow, \overline{\searrow} = \nearrow, \overline{\uparrow} = \downarrow, \overline{\downarrow} = \uparrow, \overline{\text{id}} = \text{id}\).
It remains to define \( B_{\pi_1 \cap \pi_2} = (S, \rightarrow, \iota, f) \) from \( B_{\pi_i} = (S_i, \rightarrow_i, \iota_i, f_i), \ i = 1, 2 \):

- \( S = S_1 \times S_2 \) since we essentially have a product construction
- \( \iota = (\iota_1, \iota_2) \) \( f = (f_1, f_2) \)
- synchronous transitions:
  - \((s_1, s_2) \xrightarrow{\text{dir}} (s'_1, s'_2)\) if \( s_i \xrightarrow{\text{dir}} s'_i \) for \( i = 1, 2 \)
- simulation of A2As:
  - \((s_1, s_2) \xrightarrow{A} (s'_1, s_2)\) if \( s_1 \xrightarrow{A} s'_1 \)
  - \((s_1, s_2) \xrightarrow{A} (s_1, s'_2)\) if \( s_2 \xrightarrow{A} s'_2 \)
- loops:
  - \((s_1, s_2) \xrightarrow{A(s_1, \rightarrow_1, \iota_1, s'_1)} (s'_1, s_2)\)
  - \((s_1, s_2) \xrightarrow{A(s_2, \rightarrow_2, \iota_2, s'_2)} (s_1, s'_2)\)

Hereby, \((S_i, \rightarrow_i, \iota_i, s'_i)\) is simply \( B_{\pi_i} \), but with new initial state \( s_i \) and new final state \( s'_i \), since we want to check that \( B_{\pi_i} \) can make a loop while switching from \( s_i \) to \( s'_i \).

For every node formula \( \sigma \) of ICPDL, we now have an A2A \( A_{\sigma} \) over \( \Lambda^{2k} \) such that, for all \( \tau \in L(A_{\text{cbm}}^{k-\text{sw}}) \) and all leaves \( u \) of \( \tau \), the following holds:

\[
(\tau, u) \in L(A_{\sigma}) \quad \text{iff} \quad \text{cbm}(\tau), u \models \sigma
\]

Given an A2A \( A \), we define \( L_{\text{root}}(A) = \{ t \mid (t, \text{root}(t)) \in L(A) \} \). For \( \Phi \in \text{ICPD}L \), we can now easily construct an A2A \( A_{\Phi}^{k-\text{sw}} \) such that, for all \( \tau \in L(A_{\text{cbm}}^{k-\text{sw}}) \), the following holds:

\[
\tau \in L_{\text{root}}(A_{\Phi}^{k-\text{sw}}) \quad \text{iff} \quad \text{cbm}(\tau) \models \Phi
\]

We proceed inductively:

- The A2As for \( \Phi_1 \lor \Phi_2 \) and \( \neg \Phi \) are constructed exactly like for node formulas.
- The A2A for \( E\sigma \) walks down non-deterministically from the root to some leaf and starts simulating \( A_{\sigma} \).
The size of $A^{k-sw}_Φ$ is exponential in $|Φ|$ and $k$. The doubly exponential time complexity comes from the fact that the transformation of an A2A into an NTA comes with another exponential blow up:

**Theorem 4.41 ([Var98]).** For an A2A $A$ with $n$ states and highest priority $ℓ$, one can effectively construct, in exponential time, an NTA $A'$ of size $2^{O(nℓ)}$ such that $L(A') = L(A)$.

Note that the A2A constructed for an ICPDL formula has only polynomial size when intersection is not used (or when allowing only a bounded number of nested intersections). Thus, we get the following corollary:

**Theorem 4.42.** $sw$-CPDL-ModelChecking $(A, Σ)$ is solvable in exponential time (when the bound $k$ on the split-width is encoded in unary).
In the previous chapters, we have studied distributed systems communicating via asynchronous data structures such as channels or stacks. Here, we change the communication paradigm to shared resources. This framework allows processes to communicate via shared variables or to synchronize via shared actions.

5.1 Architecture for shared resources

**Definition 5.1 (Architecture).** An architecture for shared resources is a tuple \( \mathcal{A} = (\text{Res}, \Sigma, R, W) \) where

- \( \text{Res} \) is a finite set of resources,
- \( \Sigma \) is a finite set of actions,
- \( R : \Sigma \to 2^{\text{Res}} \) gives the set of resources read by actions,
- \( W : \Sigma \to 2^{\text{Res}} \) gives the set of resources written by actions.

\[ \diamond \]

In general, we assume that \( W(a) \neq \emptyset \) for all actions \( a \in \Sigma \).

**Example 5.2 (Peterson’s mutual exclusion algorithm).** This algorithm can be described as follows:
We may also describe Peterson’s algorithm by the following automata with variables:

Example 5.3 (Processes with shared variables). Peterson’s algorithm is an example of a distributed system consisting of processes communicating exclusively with shared variables. Such systems are described with architectures satisfying:

- \( \text{Res} = \text{Procs} \cup \text{Var} \),
- \( \Sigma = \Sigma_1 \cup \Sigma_2 \) where \( \Sigma_1 \) and \( \Sigma_2 \) are given below with the read and write functions:

Each action belongs to one process. We denote by \( \Sigma_p = \{ a \in \Sigma \mid p \in W(a) \} \) the set of actions of process \( p \in \text{Procs} \). We have \( \Sigma = \biguplus_{p \in \text{Procs}} \Sigma_p \).
For Peterson’s algorithms, we have two processes with sets of actions $\Sigma_1$ and $\Sigma_2$ as described above.

In addition to reading/modifying the state of its process, an action $a$ may read shared variables in $R(a) \cap \text{Var}$ and modify shared variables in $W(a) \cap \text{Var}$. ♦

**Example 5.4 (Processes with shared actions).** Here, we have a set of processes which may synchronize by executing simultaneously some shared actions. The architecture satisfies

- $\text{Res} = \text{Procs}$,
- for all $a \in \Sigma$, $R(a) = W(a) \neq \emptyset$. ♦

Again, for each $p \in \text{Procs}$, we let $\Sigma_p = \{a \in \Sigma \mid p \in W(a)\}$ be the set of actions in which $p$ participates. We have $\Sigma = \bigcup_{p \in \text{Procs}} \Sigma_p$ but here the union needs not be disjoint. If $R(a) = \{p\}$ is a singleton, then $a$ is an action internal to process $p$. Otherwise, $a$ is a synchronization between all processes in $R(a)$.

As a special case, we have the synchronization by (binary) rendez-vous when $|R(a)| \leq 2$ for all $a \in \Sigma$. Behaviours of such systems are often depicted as follows:

![Diagram of synchronization](image)

**Example 5.5 (Cellular distributed systems).** We have again $\text{Res} = \text{Procs}$. An action may read the states of several processes, but it modifies the state of only one process. Hence, we have $|W(a)| = 1$ for all $a \in \Sigma$. The set of actions of process $p$ is $\Sigma_p = \{a \in \Sigma \mid p \in W(a)\}$. We have $\Sigma = \bigcup_{p \in \text{Procs}} \Sigma_p$.

Often, processes are organized regularly, e.g., as a line or a grid. Then an action of process $p$ reads the states of $p$’s neighbours and modifies the state of $p$. ♦

### 5.2 Asynchronous automata

**Definition 5.6.** An *Asynchronous automaton* (AsyncA) over an architecture $\mathfrak{A} = (\text{Res}, \Sigma, R, W)$ is a tuple $S = ((S_r)_{r \in \text{Res}}, (\delta_a)_{a \in \Sigma}, I, F)$ where

- $S_r$ is the finite set of local states for resource $r \in \text{Res}$,
  for $I \subseteq \text{Res}$, we let $S_I = \prod_{r \in I} S_r$,
- $\delta_a \subseteq S_{R(a)} \times S_{W(a)}$ is the set of transitions for action $a \in \Sigma$,
\( \iota \in S_{Res} \) is the global initial state,
\( F \subseteq S_{Res} \) is the set of global accepting states.

The AsyncA \( S \) is deterministic if all local transition relations \( (\delta_a)_{a \in \Sigma} \) are functional.

**Example 5.7 (Peterson’s algorithm continued).** It is an asynchronous automaton with sets of local states
\[
S_{P_1} = S_{P_2} = \{1, 2, 3, 4, 5\}, \quad S_{f_1} = S_{f_2} = \{0, 1\}, \\
S_{\text{turn}} = \{1, 2\}, \quad \text{initial global state } \iota = (1, 0, 1, 0, 1) \quad (\text{ordered } P_1, f_1, \text{turn, } f_2, P_2),
\]
and local transitions for actions of the first process:

\[
\begin{array}{c|c|c|c|c|c}
 & 1 & f_1 \leftarrow & 2 & 2 & \text{turn} \leftarrow \text{turn} \rightarrow 3 \\
\hline
P_i & 1 & 2 & 3 & 4 & 5 \\
\hline
f_1 & & f_1 \leftarrow & & & f_2 \leftarrow 0 \\
\hline
\text{turn} & & & turn \rightarrow 4 & & \\
\hline
f_2 & & & & & \\
\hline
P_i & 3 & & CS_1 & 5 & f_1 \leftarrow 0 \\
\hline
f_1 & & & & 5 & \\
\hline
\text{turn} & & & & 5 & \\
\hline
f_2 & & & & & \\
\hline
\end{array}
\]

Hence, this asynchronous automaton has 16 states and 14 transitions.

We first define the global, interleaved, sequential semantics of asynchronous automata. Then, we will see a semantics over graphs which emphasizes the concurrency/distribution of executions of AsyncA.

**Definition 5.8 (Operational semantics).** We associate with an AsyncA \( S \), a nondeterministic finite state automaton over \( \Sigma \):
\[
\mathcal{S} = (S_{Res}, \Delta, \iota, F)
\]
where the global transition relation is given by \( \overrightarrow{s} \Rightarrow \overrightarrow{s'} \) if \( (\overrightarrow{s}_{R(a)}, \overrightarrow{s}_{W(a)}) \in \delta_a \) and \( \overrightarrow{s}_{Res \setminus W(a)} = \overrightarrow{s}_{Res} \). The sequential language of \( S \) is \( L_{seq}(S) = L(\mathcal{S}) = \{w \in \Sigma^* \mid \iota \overrightarrow{\Rightarrow} F\} \).
Here, \( \overrightarrow{s} = (s_r)_{r \in \text{Res}} \) is a global state and \( \overrightarrow{s}_I = (s_r)_{r \in I} \) is its restriction to \( I \subseteq \text{Res} \).

The global automaton \( \mathcal{S} \) associated with an AsyncA \( S \) is usually huge. This phenomenon is called the combinatorial explosion due to the interleaved semantics of several distributed processes. For instance, the AsyncA of Peterson’s algorithm has 200 states (not all are reachable) and the single action \( f_1 \leftarrow 1 \) which generates a unique local transition of the AsyncA \( S \) gives rise to 40 global transitions in \( \mathcal{S} \).

**Definition 5.9 (CREW: Dependence/Independence).** Shared resources allow concurrent reads (CR) but require exclusive write (EW). This gives rise to a dependence relation on the actions of an architecture \( \mathcal{A} = (\text{Res}, \Sigma, R, W) \):
\[
D = \{(a, b) \in \Sigma^2 \mid W(a) \cap W(b) \neq \emptyset \lor W(a) \cap R(b) \neq \emptyset \lor R(a) \cap W(b) \neq \emptyset\}.
\]

The corresponding independence relation is \( I = \Sigma^2 \setminus D \).

The pair \( (\Sigma, D) \) (resp. \( (\Sigma, I) \)) is called a dependence (resp. independence) alphabet.
The three conditions used in the definition of $D$ are commonly referred to as Bernstein’s conditions [Ber66].

The dependence and the independence relations are symmetric. The dependence relation is also reflexive, assuming that $W(a) \neq \emptyset$ for all $a \in \Sigma$. A clique of $D$ is a subset $A \subseteq \Sigma$ of actions which are pairwise dependent: $A^2 \subseteq D$.

For some theorems to be applicable, such as Zielonka’s Theorem and equivalence of asynchronous automata and MSO logic, we will require that $A$ fulfills an additional property: We call $A$ symmetric if $\{(a,b) \in \Sigma \times \Sigma \mid W(a) \cap R(b) \neq \emptyset\}$ is reflexive and symmetric.

**Example 5.10 (Peterson’s algorithm continued).** The subsets $\Sigma_1$ and $\Sigma_2$ defined in Example 5.2 are dependence cliques. The additional dependences between actions of $P_1$ and actions of $P_2$ are given below.

![Diagram of Peterson's algorithm]

Note that the architecture from Example 5.2 is not symmetric. ♦

**Exercise 5.11.** Modify the model of Peterson’s algorithm in terms of a symmetric architecture.

Independent actions may be executed in any order (or even simultaneously) in an AsyncA without changing the resulting state. This is called the diamond property.

**Definition 5.12 (Diamond).** A word automaton over $\Sigma$ has the diamond property for the independence relation $I$ if for all $(a,b) \in I$ we have

- for all transitions $(s, a, s')$ and $(s', b, s'')$ there exist transitions $(s, b, s'')$ and $(s'', a, s''')$,
- for all transitions $(s, a, s')$ and $(s, b, s'')$ there exist transitions $(s', b, s''')$ and $(s'', a, s'''').

Clearly, the global automaton $\mathcal{S}$ of an asynchronous automaton $\mathcal{S}$ has the diamond property for the independence relation of Definition 5.9.
Definition 5.13 (Partially commutative monoid). Let \((\Sigma, D)\) be a dependence alphabet. The trace equivalence \(\sim\) is the least congruence over \(\Sigma^*\) such that \(ab \sim ba\) for all \((a, b) \in \Gamma\). It is also the least equivalence relation such that \(uabv \sim ubav\) for all \((a, b) \in \Gamma\) and \(u, v \in \Sigma^*\).

The quotient \(\Sigma^*/\sim\) is the partially commutative monoid, or trace monoid. An element of \(\Sigma^*/\sim\) is an equivalence class \([w]_\sim\) (or simply \([w]\)) of some word \(w\).

A word language \(L \subseteq \Sigma^*\) is trace closed if for all \(w \in L\) we have \([w] \subseteq L\). ♦

Clearly, the language accepted by an automaton having the diamond property is trace closed. We deduce

Proposition 5.14. Let \(S\) be an \(\text{AsyncA}\) over the architecture \(\mathfrak{A}\) and let \(D\) be the associated dependence relation. The language \(L_{\text{seq}}(S)\) is both regular and trace closed.

A natural question is whether the converse holds. This is a celebrated and difficult result due to Zielonka.

Theorem 5.15 (Zielonka). Let \(\mathfrak{A}\) be a symmetric architecture with associated dependence alphabet \((\Sigma, D)\). Let \(L \subseteq \Sigma^*\) be a regular language which is trace closed. There exists a deterministic \(\text{AsyncA} S\) over \(\mathfrak{A}\) such that \(L_{\text{seq}}(S) = L\).

The above theorem is effective, meaning that if \(L\) is effectively given by some finite automaton, then we can effectively construct the corresponding \(\text{AsyncA} S\).

Corollary 5.16. Assume \(\mathfrak{A}\) is symmetric. Given an \(\text{AsyncA} S\), we can effectively construct

- an equivalent deterministic \(\text{AsyncA} S_1\): \(L_{\text{seq}}(S_1) = L_{\text{seq}}(S)\).
- a deterministic \(\text{AsyncA} S_2\) for the complement: \(L_{\text{seq}}(S_2) = \Sigma^* \setminus L_{\text{seq}}(S)\).

5.3 Traces as graphs or partial orders

Instead of an equivalence class of words, which may be huge, a behaviour of an \(\text{AsyncA}\) can be described with a single graph or partial order. This graph makes explicit the causality between events (ordering) and the concurrency between events (absence of ordering).

Definition 5.17 (Traces as partial orders). Let \((\Sigma, D)\) be a dependence alphabet. A Mazurkiewicz trace over \((\Sigma, D)\) is a tuple \(t = (V, \leq, \lambda)\) where

- \(V\) is a finite set of events,
- \(\lambda: V \rightarrow \Sigma\) labels events with actions,
• \( \leq \subseteq V^2 \) is a partial order relation describing the causality between events. We let \( \preceq = \leq \setminus \leq^2 \) be the immediate successor relation. The following two conditions should be satisfied for all \( e, f \in V \):

\[
\begin{align*}
T_1 & \quad \lambda(e) D \lambda(f) \implies e \leq f \text{ or } f \leq e: \text{ dependent events must be ordered,} \\
T_2 & \quad e \preceq f \implies \lambda(e) D \lambda(f): \text{ the immediate successor relation must be justified by the dependency of events.}
\end{align*}
\]

We denote by \( \mathcal{M}(\Sigma, D) \) the set of finite traces over \( (\Sigma, D) \).

**Example 5.18 (Traces).** We usually depict the Hasse diagram \((V, \preceq, \lambda)\) of a trace \( t = (V, \leq, \lambda) \). Consider the dependence alphabet \((\Sigma, D)\) on the left below. The Hasse diagram of a trace over \((\Sigma, D)\) is depicted on the right.

We may assume that the dependence alphabet describes a systems with shared actions (cf. Example 5.4) and three processes: \( \Sigma_p = \{a, b, f\} \), \( \Sigma_q = \{b, c, d\} \) and \( \Sigma_r = \{d, e, f\} \). Then, the trace above can be described with the process-based view below.

As we will see below, equivalence classes of words or traces viewed as partial orders are equivalent objects.

**Definition 5.19 (Linearization).** Let \( t = (V, \leq, \lambda) \in \mathcal{M}(\Sigma, D) \) be a trace. A linear extension of \( \leq \) is a total order \( \sqsubseteq \) compatible with \( \leq \), i.e., \( \leq \subseteq \sqsubseteq \).

A linearization of \( t \) is a word \( w = (V, \sqsubseteq, \lambda) \) where \( \sqsubseteq \) is a linear extension of \( \leq \). Here, we identify the total order \( (V, \sqsubseteq, \lambda) \) with the word \( \lambda(e_1) \lambda(e_2) \cdots \lambda(e_n) \in \Sigma^* \) if \( V = \{e_1, e_2, \ldots, e_n\} \) and \( e_1 \sqsubseteq e_2 \sqsubseteq \cdots \sqsubseteq e_n \).

We let \( \text{Lin}(t) \subseteq \Sigma^* \) be the set of linearizations of \( t \).

**Example 5.20.** Let \( t \) be the trace depicted in Example 5.23 below. Then, \( \text{Lin}(t) \) consists of 25 words, the least in lexicographic order being \( acbabddc \) and the last being \( cddababc \).
**Proposition 5.21.** Let \( t = (V, \leq, \lambda) \in \mathcal{M}(\Sigma, D) \) be a trace. Then, \( \text{Lin}(t) \) is an equivalence class \([w]\) of some/any word \( w \in \text{Lin}(t)\). More precisely,

1. \( \forall w_1, w_2 \in \text{Lin}(t), \text{we have } w_1 \sim w_2. \)
2. \( \text{Lin}(t) \) is trace closed.

We deduce that \( \text{Lin}: \mathcal{M}(\Sigma, D) \to \Sigma^*/\sim \) is a well-defined function.

We will see below that \( \text{Lin} \) is actually a bijection from \( \mathcal{M}(\Sigma, D) \) to \( \Sigma^*/\sim \).

**Proof.** 1. Let \( w_1 = (V, \sqsubseteq_1, \lambda) \) and \( w_2 = (V, \sqsubseteq_2, \lambda) \) be linearizations of \( t \). The proof is by induction on \( |V| \). Let \( e \in V \) be maximal wrt \( \sqsubseteq_1 \) and \( a = \lambda(e) \). Then, \( w_1 = u_1a \). It is easy to check that \( t' = (V \setminus \{ e \}, \leq, \lambda) \in \mathcal{M}(\Sigma, D) \) is a trace and that \( u_1 \) is a linearization of \( t' \).

Now, let \( f \in V \) with \( e \sqsubseteq_2 f \). We show by contradiction that \( a \not\sqsubset \lambda(f) \). Indeed, if \( a \not\sqsubset \lambda(f) \) then by \((T_1)\) we get \( e < f \) or \( f < e \). The second case is impossible since \( e \sqsubseteq_2 f \) and \( \sqsubseteq_2 \) is a linear extension of \( \leq \). The first case is impossible since \( e \) is maximal for \( \sqsubseteq_1 \) which is a linear extension of \( \leq \). We deduce that \( w_2 = u_2av_2 \) with \( a \not\sqsubset v_2 \), i.e., \( a \not\sqsubset b \) for all \( b \in \text{alph}(v_2) \). Here, \( u_2 \) (resp. \( v_2 \)) is the prefix (resp. suffix) of \( w_2 \) induced by \( \{ f \in V \mid f \sqsubseteq_2 e \} \) (resp. \( \{ f \in V \mid e \sqsubseteq_2 f \} \)). We obtain \( w_2 \sim u_2v_2a \) and it is easy to check that \( u_2v_2 \in \text{Lin}(t') \). By induction \( u_1 \sim u_2v_2 \).

Hence, \( w_1 = u_1a \sim u_2v_2a \sim u_2av_2 = w_2 \).

2. Assume that \( w_1 = uabv \) and \( w_2 = vabu \) with \( w_1 \in \text{Lin}(t) \) and \( a \sqsubset b \). We have to show that \( w_2 \in \text{Lin}(t) \). Let \( e, f \in V \) be the events corresponding to these occurrences of \( a \) and \( b \). We show that \( e \parallel f \) are concurrent in \( t \), i.e., \( (e \not\sqsubset f \lor f \not\sqsubset e) \). First, assume that \( e < f \). Since \( f \) is a successor of \( e \) in \( w_1 \) we get \( e < f \), which implies \( a \sqsubset b \) by \((T_2)\), a contradiction. Second, \( f < e \) is a contradiction with \( e \sqsubseteq_1 f \) and \( \sqsubseteq_1 \) is a linear extension of \( \leq \).

We deduce that \( \sqsubseteq_2 = \sqsubseteq_1 \setminus \{(e, f)\} \cup \{(f, e)\} \) is a linear extension of \( \leq \). Therefore, \( w_2 = (V, \sqsubseteq_2, \lambda) \in \text{Lin}(t) \).

To prove that the map \( \text{Lin} \) from traces to equivalence classes is a bijection, we construct the inverse map. We show how to associate with every word \( w \in \Sigma^* \) a trace \( \text{Tr}(w) \in \mathcal{M}(\Sigma, D) \).

**Definition 5.22 (Words to traces).** Let \( w \in \Sigma^* \) be a word. We define the trace \( \text{Tr}(w) = (V_w, \leq_w, \lambda) \in \mathcal{M}(\Sigma, D) \) as follows:

1. \( V_w = \{(a, i) \mid a \in \Sigma \land 1 \leq i \leq |w|_a \} \) the set of action occurrences in \( w \),
   a pair \((a, i)\) stands for the \( i \)th occurrence of \( a \) in \( w \),
   \(|w|_a\) is the number of occurrences of \( a \) in \( w \),
   we denote by \( \sqsubseteq_w \) the total order induced by \( w \) on \( V \):
   \((a, i) \sqsubseteq_w (b, j) \) if the \( i \)th occurrence of \( a \) is before the \( j \)th occurrence of \( b \) in \( w \),
2. \( \lambda((a, i)) = a \),

73
3. $\leq_w = (E_w)^+$ is the transitive closure of the relation

$$E_w = \{((a, i), (b, j)) \mid a \mathcal{D} b \land (a, i) \subseteq_w (b, j)\}.$$ 

**Example 5.23.** Consider the dependence alphabet depicted on the left below. Then, the trace $\text{Tr}(acbdabdc)$ is depicted on the right.

**Proposition 5.24 (Words to traces).** The following holds.

1. $\forall w \in \Sigma^*, \text{Tr}(w) \in \mathcal{M}(\Sigma, D)$ is indeed a trace and $w \in \text{Lin}(\text{Tr}(w))$.
2. $\forall w_1, w_2 \in \Sigma^*$, we have $w_1 \sim w_2$ iff $\text{Tr}(w_1) = \text{Tr}(w_2)$.
3. $\forall t \in \mathcal{M}(\Sigma, D), \forall w \in \text{Lin}(t)$, we have $t = \text{Tr}(w)$.

Therefore, the diagram above commutes and the maps $\text{Tr}$ and $\text{Lin}$ are inverse bijections between traces seen as equivalence classes and traces seen as partial orders.

**Proof.** 1. The relation $E_w$ is antisymmetric: $(e, f), (f, e) \in E_w$ implies $e \subseteq f$ and $f \subseteq e$ which implies $e = f$. Hence $\leq_w$ is a partial order. We check that conditions (T1) and (T2) are satisfied. First, assume that $\lambda(e) \mathcal{D} \lambda(f)$. Since $\subseteq$ is a total order, either $e \subseteq f$ and we get $(e, f) \in E_w$, or $f \subseteq e$ and $(f, e) \in E_w$. We deduce that (T1) holds. Second, if $e \prec f$ then $(e, f) \in E_w$ and we get $\lambda(e) \mathcal{D} \lambda(f)$. Therefore, $\text{Tr}(w)$ is indeed a trace. Moreover, $\leq = E_w^+ \subseteq_w = \subseteq_w$ and $w \in \text{Lin}(\text{Tr}(w))$.

2. Assume $w_1 = uabv$ and $w_2 = ubav$ with $u, v \in \Sigma^*$ and $a \not\mathcal{D} b$. Then $V_{w_1} = V_{w_2}$ and $E_{w_1} = E_{w_2}$. Therefore, $\text{Tr}(w_1) = \text{Tr}(w_2)$.

Conversely, let $w_1, w_2 \in \Sigma^*$ with $\text{Tr}(w_1) = \text{Tr}(w_2) = t$. By (1) we get $w_1, w_2 \in \text{Lin}(t)$ and by Proposition 5.21 we deduce $w_1 \sim w_2$.

3. Let $t = (V, \leq, \lambda) \in \mathcal{M}(\Sigma, D)$ and $w = (V, \subseteq, \lambda) \in \text{Lin}(t)$ be a linearization of $t$. We show below that $\subseteq \subseteq E_w \subseteq \leq$. We deduce that $\leq = E_w^+$ and $t = \text{Tr}(w)$.

First, assume that $e \prec f$, then $e \prec f$ and $\lambda(e) \mathcal{D} \lambda(f)$ by (T2). Since $\subseteq$ is a linear extension of $\leq$ we get $e \subseteq f$. Hence, $(e, f) \in E_w$. Second, assume that $(e, f) \in E_w$. Then, $e \subseteq f$ and $\lambda(e) \mathcal{D} \lambda(f)$. By (T1) we get $e \leq f$ or $f < e$. The second case contradicts $e \subseteq f$. Hence, $e \leq f$. 

74
Actually, $\Sigma^*/\sim$ and $\mathbb{M}(\Sigma, D)$ are isomorphic monoids. Since $\sim$ is a congruence, the quotient $\Sigma^*/\sim$ of the free monoid is also a monoid. On the other hand, it is easy to define a concatenation on $\mathbb{M}(\Sigma, D)$. Let $t_1 = (V_1, \leq_1, \lambda_1)$ and $t_2 = (V_2, \leq_2, \lambda_2)$ be two traces. Wlog, we assume that $V_1 \cap V_2 = \emptyset$. The concatenation $t_1 \cdot t_2$ is $t = (V_1 \cup V_2, \leq, \lambda_1 \cup \lambda_2)$ where $\leq$ is the transitive closure of $\leq_1 \cup \leq_2 \cup \{(e, f) \in V_1 \times V_2 \mid \lambda_1(e) \mathrel{D} \lambda_2(f)\}$. We can check that $t$ is indeed a trace. The concatenation is associative and the unit element is the empty trace. Therefore, $(\mathbb{M}(\Sigma, D), \cdot)$ is a monoid. Moreover, the maps $\text{Tr}$ and $\text{Lin}$ are morphisms.

We show now that runs of AsyncA can be defined directly on traces seen as partial orders. First, we introduce some notations. Let $t = (V, \leq, \lambda) \in \mathbb{M}(\Sigma, D)$ be a trace over $\mathfrak{A} = (\text{Res}, \Sigma, R, W)$. For all $r \in \text{Res}$, we let $V_r = \{e \in V \mid r \in W(a)\}$ be the set of events that write on resource $r$. For an event $e \in V$, we denote by $\lhd e = \{f \in V \mid f < e\}$ its strict past.

**Definition 5.25 (Semantics of AsyncA over traces).**
Consider an AsyncA $S = ((S_r)_{r \in \text{Res}}, (\delta_a)_{a \in \Sigma}, t, F)$ over $\mathfrak{A} = (\text{Res}, \Sigma, R, W)$. Let $t = (V, \leq, \lambda) \in \mathbb{M}(\Sigma, D)$ be a trace over the associated dependence alphabet. A run of $S$ over $t$ is a map $\rho: V \to \bigcup_{I \subseteq \text{Res}} S_I$ such that for all $e \in V$ with $a = \lambda(e)$, we have

- **R$_1$** $\rho(e) \in S_{W(a)}$: when $S$ executes $e$ it writes $\rho(e)(r)$ on resources $r \in W(a)$.
- **R$_2$** $(\rho^-(e), \rho(e)) \in \delta_a$ is a local transition, where $\rho^-(e) \in S_{R(a)}$ is defined for all $r \in R(a)$ by

  $$\rho^-(e)(r) = \begin{cases} \iota(r) & \text{if } \lhd e \cap V_r = \emptyset \\ \rho(\max(\lhd e \cap V_r))(r) & \text{otherwise.} \end{cases}$$

The run $\rho$ is accepting if $\text{Last}(\rho) \in F$, where $\text{Last}(\rho) \in S_{\text{Res}}$ is defined for all $r \in \text{Res}$ by

$$\text{Last}(\rho)(r) = \begin{cases} \iota(r) & \text{if } V_r = \emptyset \\ \rho(\max(V_r))(r) & \text{otherwise.} \end{cases}$$

The trace language accepted by $S$ is

$$L(S) = \{t \in \mathbb{M}(\Sigma, D) \mid S \text{ admits an accepting run on } t\}.$$

**Example 5.26 (Peterson’s algorithm continued).** Consider the AsyncA $S$ for Peterson’s algorithm defined in Example 5.7. A run of $S$ over a trace is given in Figure 5.1.

**Proposition 5.27 (Sequential vs. trace semantics).** Consider an AsyncA $S = ((S_r)_{r \in \text{Res}}, (\delta_a)_{a \in \Sigma}, t, F)$ over the architecture $\mathfrak{A} = (\text{Res}, \Sigma, R, W)$. The following holds:

1. $\forall w \in L_{\text{seq}}(S) = L(\overline{S})$, we have $\text{Tr}(w) \in L(S)$.
2. $\forall t \in L(S)$, we have $\text{Lin}(t) \subseteq L_{\text{seq}}(S) = L(\overline{S})$.

**Exercise 5.28.** Prove Proposition 5.27.
Figure 5.1: A run of Peterson's AsyncA over a trace
5.4 Checking Closure of DFAs

In view of Zielonka’s theorem, it is natural to ask whether a given regular language is trace closed:

**Closure**:  
Instance: \( \mathbb{A} = (\mathbb{R}, \Sigma, \mathbb{R}, \mathbb{W}) \) (or \((\Sigma, \mathbb{D})\)); DFA \( \mathcal{B} \) over \( \Sigma \)  
Question: Is \( L(\mathcal{B}) \) trace closed?

**Theorem 5.29.** Closure is decidable in polynomial time.

**Proof.** We minimize \( \mathcal{B} \) to obtain the DFA \( \mathcal{B}_{\text{min}} = (Q, \delta, \iota, F) \), and then simply check whether \( \mathcal{B}_{\text{min}} \) has the diamond property (cf. Definition 5.12). Both can be done in polynomial time. Thus, it remains to show that \( L(\mathcal{B}_{\text{min}}) \) is trace closed iff, for all \( q \in Q \) and \( (a, b) \in I \), we have \( \delta(q, ab) = \delta(q, ba) \).

The direction \( \Leftarrow \) is obvious.

Towards \( \Rightarrow \), suppose \( q \in Q \) and \( (a, b) \in I \) such that \( \delta(q, ab) \neq \delta(q, ba) \). Then, we can find \( u, v \in \Sigma^* \) such that

- \( \delta(\iota, u) = q \) and
- \( \delta(q, abv) \in F \iff \delta(q, abv) \notin F \).

Thus, \( uabv \in L(\mathcal{B}_{\text{min}}) \) iff \( uabv \notin L(\mathcal{B}_{\text{min}}) \), which implies that \( L(\mathcal{B}_{\text{min}}) \) is not trace closed. \( \blacksquare \)

Alternatively, one may consider \( \mathcal{B} \) as a partial specification and be interested in the following problem:

**Reg-Closure(\( \mathbb{A} \)):**  
Instance: DFA \( \mathcal{B} \) over \( \Sigma \)  
Question: Is \([L(\mathcal{B})] \) regular?

Unfortunately, this problem is, in general, undecidable:

**Theorem 5.30 ([Sak92]).** Reg-Closure is decidable iff the relation \( I \) is transitive.

5.5 Star-Connected Regular Expressions

There is a class of regular (word) expressions whose closure is guaranteed to be regular:
Definition 5.31. A regular expression $\alpha$ over $\Sigma$ is star-connected if, for all subexpressions $\beta^*$ of $\alpha$, $\text{Tr}(L(\beta))$ consists only of connected traces. ◊

Theorem 5.32 ([Och95]). Suppose $\mathfrak{A}$ is symmetric. For $L \subseteq \mathbb{M}(\Sigma, D)$, the following statements are equivalent:

- There is an AsyncA $S$ such that $L(S) = L$.
- There is a star-connected regular expression $\alpha$ over $\Sigma$ such that $\text{Tr}(L(\alpha)) = L$.

Example 5.33. Suppose $D$ is given by $a - b - c$.

- $(ac)^*$ is not star-connected.
- $(a + c + (acb)^*)^*$ is star-connected.
- $(acb^*)^*$ is not star-connected. ◊

5.6 Logic Specifications

Traces come with a canonical MSO logic. The set $\text{MSO}(\mathfrak{A})$ of formulas from monadic second-order logic is given by the grammar:

$$\varphi ::= a(x) \mid x \leq y \mid x \in X \mid \varphi \lor \varphi \mid \neg \varphi \mid \exists x \varphi \mid \exists X \varphi$$

where $a \in \Sigma$. As usual, for a sentence $\varphi \in \text{MSO}(\mathfrak{A})$, we let $L(\varphi) = \{ t \in \mathbb{M}(\Sigma, D) \mid t \models \varphi \}$.

Example 5.34. In MSO logic, the desired specification of Peterson’s mutual exclusion protocol (cf. Example 5.2) reads as follows:

$$\neg \exists x \exists y (x \parallel y \land \text{CS}_1(x) \land \text{CS}_2(y))$$

where $x \parallel y$ is defined as $\neg (x \leq y \lor y \leq x)$. ◊

It turns out that AsyncA and MSO logic are expressively equivalent:

Theorem 5.35 ([Tho90]). Let $\mathfrak{A}$ be symmetric. For $L \subseteq \mathbb{M}(\Sigma, D)$, the following statements are equivalent:

- There is an AsyncA $S$ over $\mathfrak{A}$ such that $L(S) = L$.
- There is a sentence $\varphi \in \text{MSO}(\mathfrak{A})$ such that $L(\varphi) = L$.

Proof. The transformation of an automaton into a formula is standard (cf. proof of Theorem 3.4, for example) and left as an exercise. For the converse translation, we proceed by induction. We will restrict to the most interesting cases: $x \leq y$ and $\neg \varphi$. 78
For $x \leq y$, we build an AsyncA $S_{x \leq y}$ accepting traces with two events $e, f$ marked as $x, y$, respectively, such that $e \leq f$. To take into account the additional marking, we extend the given architecture $\mathfrak{A} = (\text{Res}, \Sigma, R, W)$ to $\mathfrak{A}' = (\text{Res}, \Sigma', R', W')$ where $\Sigma' = \Sigma \times \{0, 1\}^2$, $W'((a, i, j)) = W(a)$, and $R'((a, i, j)) = R(a)$. The idea is that a label of the form $(a, 1, j)$ indicates that the event is marked $x$, and a label of the form $(a, i, 1)$ indicates that the event is marked $y$.

Note that this gives rise to a new independence relation $D' \subseteq \Sigma' \times \Sigma'$, and we have $(a, i, j) D' (a', i', j')$ iff $a D a'$.

We are looking for an AsyncA $S_{x \leq y}$ for the trace language $L_{x \leq y}$, which is defined to be the set of traces $(V, \leq, \lambda) \in \mathcal{M}(\Sigma', D')$ such that there are $e, f \in V$ satisfying

- $e \leq f$,
- for all $g \in V$, $\lambda(g) \in \Sigma \times \{1\} \times \{0, 1\}$ iff $g = e$, and
- for all $g \in V$, $\lambda(g) \in \Sigma \times \{0, 1\} \times \{1\}$ iff $g = f$.

Instead of constructing $S_{x \leq y}$ directly, we will build an NFA $B$ over $\Sigma'$ such that $L(B) = \text{Lin}(L_{x \leq y})$. Zielonka’s theorem will then yield the desired AsyncA.

The NFA $B$ should accept a word $w \in (\Sigma')^*$ iff

- either it is of the form $w = u(a, 1, 1)v$ with $u, v \in (\Sigma \times \{0\})^*$,
- or it is of the form $w = u(a, 1, 0)v(b, 0, 1)v'$ such that $u, v, v' \in (\Sigma \times \{0\})^*$ and $v$ is of the form $v_1(a_1, 0, 0) \ldots v_n(a_n, 0, 0)$ with $a D a_1 D a_2 D \ldots D a_n D b$.

Note that such a $B$ can be easily found. It essentially guesses the next letter $a_{i+1}$ until some letter is found that is dependent on $b$.

From $B$, using Zielonka’s theorem, we obtain an AsyncA $S_{x \leq y}$ such that $L(S_{x \leq y}) = \text{Tr}(L(B)) = L_{x \leq y}$.

For $\neg \varphi$, we assume that we are given an AsyncA $S_{\varphi}$ such that $L(S_{\varphi}) = L(\varphi)$. Then, $L = \text{Lin}(L(S_{\varphi}))$ is regular and trace closed. Thus, $\Sigma^* \setminus L$ is regular and trace closed, too. By Zielonka’s theorem, there exists an AsyncA $S_{\neg \varphi}$ such that $L(S_{\neg \varphi}) = \text{Tr}(\Sigma^* \setminus L) = L(\neg \varphi)$.

It immediately follows that MSO model checking of asynchronous automata (over a symmetric architecture $\mathfrak{A}$) is decidable:

<table>
<thead>
<tr>
<th>MSO-ModelChecking($\mathfrak{A}$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance: AsyncA $S$ over $\mathfrak{A}$; $\varphi \in \text{MSO}(\mathfrak{A})$</td>
</tr>
<tr>
<td>Question: $L(S) \subseteq L(\varphi)$?</td>
</tr>
</tbody>
</table>


