Product automaton, Parallel Composition, Hiding

1 The Product Automaton

In practical examples, one might prefer a modular way of composing automata, rather than defining large automata monolytically. To this end, we define a first notion of product automata.

Let $M_1 = (Q_1, \Sigma_1, \delta_1, r_0, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, s_0, F_2)$ be two deterministic automata. The *product-automaton* $M_1 \times M_2$ is defined as follows:

$$M_1 \times M_2 = (Q_1 \times Q_2, \Sigma_1 \cap \Sigma_2, \delta, (r_0, s_0), F_1 \times F_2),$$

where for $(q_1, q_2) \in Q_1 \times Q_2$ and $x \in \Sigma_1 \cap \Sigma_2$ we have

$$\delta((q_1, q_2), x) = (\delta_1(q_1, x), \delta_2(q_2, x)).$$

So the states of the product automaton are pairs of original states of the form (q_1, q_2). Informally, the product automaton can be viewed as the combination of two automata that operate in *lock-step*: The M_1-component proceeds for some symbol $a \in \Sigma$ exactly when the M_2-component does so.

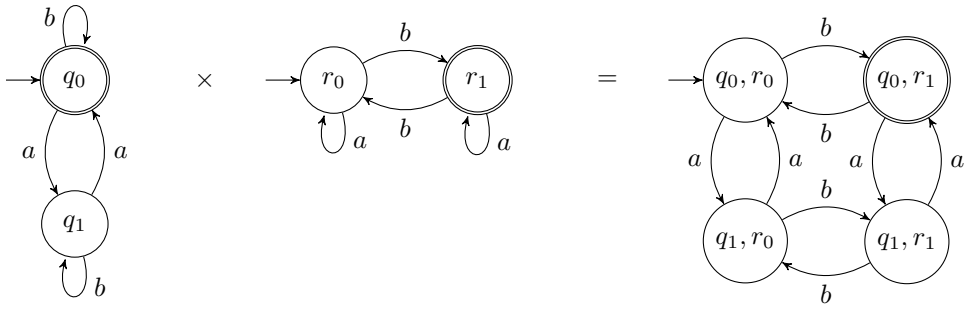

Figure 1: De product-automaat $M_1 \times M_2$.

The state of the product automaton is a pair (q_i, q_j), with $q_i \in Q_1$ and $q_j \in Q_2$. The automaton is in an accepting state (q_i, q_j) if both $q_i \in F_1$ and $q_j \in F_2$. So the set of final states is $F_1 \times F_2$. It will be clear (informally) that a word w moves the product-automaton into an accepting state, if and only if both M_1 and M_2 proceed to an accepting state by reading w. Hence, $w \in L(M_1 \times M_2)$ if and only if $w \in L(M_1)$ and $w \in L(M_2)$. In other words: $L(M_1 \times M_2) = L(M_1) \cap L(M_2)$. In [Sudkamp], Section 6.4 ("Closure properties of Regular Languages") it has been shown that the intersection of two regular languages is again regular. The product automaton provides an alternative method to demonstrate this *directly*: we have just constructed an automaton that accepts the intersection.

Exercise. Assume that we define an automaton similar to the product automaton, but this time we make a different choice for the set of accepting states, viz.: $(F_1 \times Q_2) \cup (Q_1 \times F_2)$. What language is recognised by this new automaton?

We now extend the definition of product-automaton to *non-deterministic* automata. The main difference is that the transition functions now yield *sets* of states, instead of single states.

$$M = M_1 \times M_2 = (Q_1 \times Q_2, \Sigma_1 \cap \Sigma_2, \delta, (r_0, s_0), F_1 \times F_2),$$

where the transition function δ is defined as follows:

$$\delta((q_1, q_2), x) = \{(q_1', q_2') \mid q_1' \in \delta_1(q_1, x)\ , \quad q_2' \in \delta_2(q_2, x)\}, \text{ if } x \neq \lambda$$
$$\delta((q_1, q_2), \lambda) = \{(q_1', q_2) \mid q_1' \in \delta_1(q_1, \lambda)\} \cup \{(q_1, q_2') \mid q_2' \in \delta_2(q_2, \lambda)\}.$$

With this definition we still obtain the same result as above: $L(M_1 \times M_2) = L(M_1) \cap L(M_2)$.

Note that the symbols in $\Sigma_1 - \Sigma_2$ or in $\Sigma_2 - \Sigma_1$ do not occur in the alphabet of the product. The reason is that they would be excluded by the "lockstep"-execution mechanism anyway. Also, note that the "lockstep mode" of execution is only applicable to symbols from $\Sigma_1 \cap \Sigma_2$, but *not* for λ-transitions: For instance, M_1 can execute a λ-transition, while M_2 doesn't change state, and vice versa. Why is this? The reason for this design choice is that λ-transitions are not viewed as communicating actions, but rather as *internal* steps. Assume that we have an NFA-λ-automaton M. Consider some a-transition of M, and now change M, so this transition is preceded and followed by a λ-transition. The modified automaton is called M' (see Figure below).

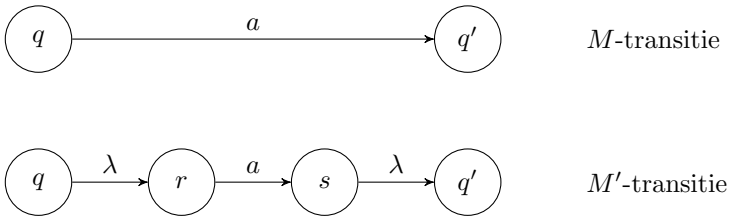

Figure 2: Adding extra λ-transitions around an existing a-transition.

Question: does this modification also change the *language* of the automaton? You can check for yourself, giving definitions from [Sudkamp], that the language is not changed: $L(M) = L(M')$. Since M and M' accept the same language, we call these automata *trace-equivalent*. Now we would expect that trace-equivalence is preserved when we put M and M' into a common context, that is, if we take the product of M or M' with an arbitrary other NFA-λ-automaton A. So, we are expecting that for any A, the products $M \times A$ and $M' \times A$ are trace-equivalent. However, choose as special case: $A = M$. It is immediately clear that $M \times M$ accepts the same language as M: $L(M \times M) = L(M)$. So we expect that $L(M' \times M)$ accepts the same language $L(M)$. Now here is where the details about the definition of λ-steps of the product automaton matter. Let us consider two alternatives:

1. *Assume* that we had treated the definition of λ-transitions similar to alphabet symbols. In that case, the product automaton would take a λ-transition if *both* components of the product would take a λ-transition. In our example this would be problematic: when M would take the a-transition, M' needs to take the λ-transition first, but this is blocked. So with this alternative definition, $M \times M$ and $M' \times M$ would not be trace-equivalent in general, contrary to our expectations.

2. Now, consider the *correct* definition, as introduced above. When M gets in the state where an a-transition is enabled, M' would get in a state where only the preceding λ-transition is possible. So the a-transition cannot yet be taken. However, according to the correct definition, $M' \times M$, can take the λ-transition, executed only by M' and leaving M unchanged. *Next*, the synchronised a-transition is possible in M and M'. This example demonstrates how $M' \times M$ would be trace-equivalent, and motivates the definition in which λ plays a special role.

The considerations above are essential to allow compositional reasoning over processes: The preservation of equivalence under context enables replacing a black-box component in a complex system by another black-box component, as long as the two components are equivalent.

Also, in order to facilitate abstract reasoning, we often want to reason about abstract versions of real processes. One way of abstraction is "information hiding". In the context of automata, information hiding can be achieved by "ignoring certain alphabet symbols", and ignoring symbols

can be interpreted formally by renaming these symbols to λ. Viewing λ-steps as abstractions of complicated processes that are hidden is another motivation why synchronisation of λ-steps is not a good idea.

Finally, note that if we consider the "input transition function" of the product automaton, as defined in [Sudkamp], Definition 5.6.2, one realises that the definition of the product automaton is quite natural: We can derive the input-transition function t of the product automaton *directly* from the input-transition functions t_1 and t_2 of M_1 and M_2. Thanks to the correct definition for λ-transitions, this looks simply as follows:

$$t((q_1, q_2), a) = \{(q_1', q_2') \mid q_1' \in t_1(q_1, a)\ , \quad q_2' \in t_2(q_2, a)\}.$$

2 Projections

We will now define the *projection* on (or: restriction to) a smaller alphabet. For alphabets Σ_1, Σ_2 with $\Sigma_2 \subseteq \Sigma_1$, the projection on alphabet Σ_2 is a function from $\Sigma_1^\star$ to $\Sigma_2^\star$. We denote this by $w \restriction \Sigma_2$. The definition is as follows (here $a \in \Sigma_1$ and $v, w \in \Sigma_1^\star$):

$$\begin{aligned} a \restriction \Sigma_2 &= a, \text{ if } a \in \Sigma_2 \\ a \restriction \Sigma_2 &= \lambda, \text{ if } a \in \Sigma_1 - \Sigma_2 \\ \lambda \restriction \Sigma_2 &= \lambda \\ (v \cdot w) \restriction \Sigma_2 &= (v \restriction \Sigma_2) \cdot (w \restriction \Sigma_2)\ . \end{aligned}$$

Informally: $w \restriction \Sigma_2$ is the word w restricted to symbols in Σ_2; in other words, with all symbols outside Σ_2 removed. As an example:

$$\begin{aligned} &(ABACB) \restriction \{A, C\} \\ &\quad = (A \cdot B \cdot A \cdot C \cdot B) \restriction \{A, C\} \\ &\quad = (A \restriction \{A, C\}) \cdot (B \restriction \{A, C\}) \cdot (A \restriction \{A, C\}) \cdot (C \restriction \{A, C\}) \cdot (B \restriction \{A, C\}) \\ &\quad = A \cdot \lambda \cdot A \cdot C \cdot \lambda = AAC\ . \end{aligned}$$

Note that we can lift the projection operator from words to languages (sets of words) as follows:

$$L \restriction \Sigma_2 = \{w \restriction \Sigma_2 \mid w \in L\}$$

Next, we lift projection to automata.

Theorem 1 *Let L be the language accepted by an NFA-λ $M_1 = (Q, \Sigma_1, \delta, q_0, F)$ and let $\Sigma_2 \subseteq \Sigma_1$. Then the language $L \restriction \Sigma_2$ will be accepted by an automaton M_2, which is denoted by $M \restriction \Sigma_2$. This operation on automata is defined as follows:*

$$M \restriction \Sigma_2 = (Q, \Sigma_2, \delta', q_0, F),$$

where δ' equals δ, except that for all $a \in \Sigma_1 - \Sigma_2$, the a-transitions are replaced by λ-transitions. More formally:

- $\delta'(q, a) = \delta(q, a)$, *for* $a \in \Sigma_2$,
- $\delta'(q, a) = \varnothing$, *for* $a \in \Sigma_1 - \Sigma_2$,
- $\delta'(q, \lambda) = \delta(q, \lambda) \cup \{q' \mid \exists a \in \Sigma_1 - \Sigma_2\,.\, q' \in \delta(q, a)\}$.

Proof (sketch). Let word $w' \in \Sigma_2^\star$ be accepted by $M_1 \restriction \Sigma_2$. Then there is a computation of the form

$$q_0 \xrightarrow{x_0} q_1 \xrightarrow{x_1} q_2 \cdots \xrightarrow{x_{n-1}} q_n,$$

with $q_n \in F$, with $w' = x_0 \cdot x_1 \cdots x_{n-1}$. Some of these x_i-labels could be λ, and some might originate from $(\Sigma_1 - \Sigma_2)$-symbols that are replaced by λ, during the construction of M_2 from M_1. Replace (in w') the latter λ's by the original symbols in $\Sigma_1 - \Sigma_2$. This yields a word over Σ_1, say w, with the following properties: (1) $w \restriction \Sigma_2 = w'$, and (2) there is a computation of M_1 for w leading from q_0 to q_n. Consequently, $w \in L(M_1)$. So for every $w' \in L(M_1 \restriction \Sigma_2)$ there is a $w \in L(M_1)$ with $w' = w \restriction \Sigma_2$. This is exactly the definition of $L(M_1) \restriction \Sigma_2$.

3 The "hiding"-operation

Let L be a language with alphabet Σ and $\Sigma' \subseteq \Sigma$. With the *projection* operation on Σ' that we just defined, we explicitly mention the symbols that should remain. Sometimes, it is more convenient to explicitly mention the symbols that we want to hide. To this end, we introduce the *hiding*-operator: $w\backslash\Sigma_1$, as an abbreviation of the projection $w \restriction (\Sigma - \Sigma_1)$. So when "hiding" the Σ_1-symbols in w, we replace all Σ_1-labels occurring in w by λ, effectively removing them. Analogously, we define the hiding-operation on *languages*: $L\backslash\Sigma_1 = \{w\backslash\Sigma_1 \mid w \in L\}$. Finally, for automaton M, we define the hiding operation $M\backslash\Sigma_1$ by replacing all symbols from Σ_1 that occur in transitions of M by λ. As above, we obtain that if M accepts L, then $M\backslash\Sigma_1$ accepts $L\backslash\Sigma_1$.

Note that even when M is a DFA or NFA, the result of the hiding-operation will always be an NFA-λ-automaton. Therefore, in practice on often transforms this intermediate result into a deterministic automaton, and minimises it using the algorithm from Section 5.7 [Sudkamp]. Remember that the minimisation algorithm only works for (complete) deterministic automata.

4 Parallel Composition

For the practice of system and software specifications, the product operation that we have introduced before is still not practical. We will rectify that in this section, by introducing the parallel composition. The main problem is that when combining automata with the product operation we are actually forced to use *identical* alphabets. Otherwise, only the intersection of the two alphabets remains after the product operation, cutting away part of the behaviour of components. The intuition behind the *parallelle* composition of automata $M_1, M_2, \ldots, M_n$ is as follows.

- When a number of components M_i have *common* alphabet symbols a, those a-transitions can only happen iff *all* these components agree to execute an a-transition *simultaneously*. This is comparable to the "lock-step mode" of product automata, and such steps are called *synchronisation steps*.
- But when some symbol, say a, does *not* occur in the alphabet of some component, say M_i, then M_i cannot block the execution of a-transitions. In particular, if an a-transition happens, the state of M_i will not change. This is different from the situation in a product automaton, where such a-transitions would not be possible, since M_i doesn't agree on them.

The definition of parallel composition is as follows. Assume $M_1 = (Q_1, \Sigma_1, \delta_1, r_0, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, s_0, F_2)$ are non-deterministic automata. Then $M_1 \parallel M_2$ is defined to be the automaton:

$$M_1 \parallel M_2 = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (r_0, s_0), F_1 \times F_2),$$

where transition function δ is defined by:

- $\delta((q_1, q_2), x) = \{(q_1', q_2') \mid q_1' \in \delta_1(q_1, x), q_2' \in \delta_2(q_2, x)\}$, if $x \neq \lambda$ and $x \in \Sigma_1 \cap \Sigma_2$
- $\delta((q_1, q_2), x) = \{(q_1', q_2) \mid q_1' \in \delta_1(q_1, x)\}$, if $x \neq \lambda$ and $x \in \Sigma_1 - \Sigma_2$
- $\delta((q_1, q_2), x) = \{(q_1, q_2') \mid q_2' \in \delta_2(q_2, x)\}$, if $x \neq \lambda$ and $x \in \Sigma_2 - \Sigma_1$
- $\delta((q_1, q_2), \lambda) = \{(q_1', q_2) \mid q_1' \in \delta_1(q_1, \lambda\} \cup \{(q_1, q_2') \mid q_2' \in \delta_2(q_2, \lambda)\}$.

It is now obvious that the parallel composition $M_1 \parallel M_2$ resembles the product $M_1 \times M_2$, but the δ-function is more complicated. Informally, we can read the definition as follows: when a word w is processed by a parallel automaton, then each parallel component works effectively on the *projection* of w on their own alphabet Σ_i. Indeed, symbols outside the alphabet Σ_i are ignored: they are not blocked by M_i, and they don't lead to a state change in M_i.

The difference between $M_1 \parallel M_2$ and $M_1 \times M_2$ can be summarised as follows:

- A word w is accepted by $M_1 \times M_2$ if it is accepted by *both* M_1 *as well as* M_2. (This restricts the alphabet of $M_1 \times M_2$ automatically to $\Sigma_1 \cap \Sigma_2$.)
- A word w is accepted by $M_1 \parallel M_2$ if both *the projection of* w *onto* Σ_1 is accepted by M_1 and also the *projection onto* Σ_2 is accepted by M_2. Hence, the alphabet of $M_1 \parallel M_2$ should be $\Sigma_1 \cup \Sigma_2$.

5 Final words on Product-automaton, Parallel Composition and Hiding

The parallel composition of $M_1 \parallel M_2$ is different from the product $M_1 \times M_2$, but there is a connection with the product and project operation. The goal of this section is to show how the product-automaton operation can be "reused" to obtain the parallel composition.

The idea is as follows. If we take the product of two automata M_1 and M_2 with different alphabets Σ_1 and Σ_2 directly, symbols outside the intersection $\Sigma_1 \cap \Sigma_2$ will be removed. We "correct" this situation, by first transforming M_1 and M_2 to automata M_1' and M_2' with the same alphabet $\Sigma_1 \cup \Sigma_2$, and only then take their product.

For M_1' we copy automaton M_1 as far as symbols in Σ_1 are concerned, but we add the symbols from $\Sigma_2 - \Sigma_1$ as extra transitions: For each $x \in \Sigma_2 - \Sigma_1$ and for each state q of M_1', we add the transition $\delta_1'(q, x) = q$. Informally, we add an x-self-loop for each state of M_1'. Now, what is the effect on M_1' if we execute a word w over the alphabet $\Sigma_1 \cup \Sigma_2$? For transitions caused by Σ_1-symbols, M_1' behaves identically to M_1. For the other symbols, we observe:

- M_1' will never block those other transitions: $\Sigma_2 - \Sigma_1$-transitions are possible from all states.
- M_1' will not change state by these transitions, since they form self-loops. So M_1' will always be in the same state as M_1 during the execution of the projected input $w \restriction \Sigma_1$.

Now consider the product $M_1' \times M_2'$, executing the word w. The fact that M_1' is in the same state as M_1 would be on input $w \restriction \Sigma_1$ *remains valid*. Also M_2' is in the same state after w as M_2' would be on $w \restriction \Sigma_2$. We conclude that $M_1' \times M_2'$ behaves exactly as $M_1 \parallel M_2$.

Example. Let M_{abc} be an automaton over alphabet $\{a, b, c\}$, for the language $L_{abc} = (abc)^\star$. We want to compose this in parallel with an automaton M_{bdc} for language $L_{bdc} = (bdc)^\star$. To do so, we first construct from M_{abc} a new automaton M'_{abc}, by adding transitions of the form $\delta'_{abc}(q_i, d) = \{q_i\}$, for all states q_i of M_{abc}. These added transitions are visible in the figure as the dashed "d-loops". Similarly, we transform M_{bdc} into automaton M'_{bdc}, by adding "a-loops" of the form $\delta'_{bdc}(r_i, a) = \{r_i\}$. Note that indeed M'_{abc} accepts word w over alphabet $\{a, b, c, d\}$ if and only if $w \restriction \{a, b, c\}$ matches $(abc)^\star$, so indeed, if the projection of w on the alphabet of M_{abc} is accepted by this automaton M_{abc}.

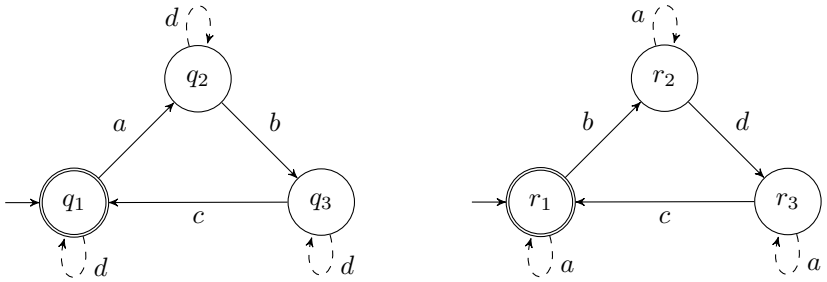

Figure 3: M'_{abc} and M'_{bdc} over the alphabet $\{a, b, c, d\}$.

When we construct the product $M'_{abc} \times M'_{bdc}$, we obtain the automaton of Figure 4.

Note that the result automaton has nine states, of which five states are unreachable. These states can be removed, without changing the accepted language. The resulting automaton has only four states, see Figure 5, and apparently accepts the language $L_{abdc} = (abdc)^\star$. Indeed, note that $w \in L_{abdc}$, if and only if both $w \restriction \{a, b, c\} \in L_{abc}$ as well as $w \restriction \{b, c, d\} \in L_{bdc}$.

In summary, we have reconstructed the parallel composition $M_1 \parallel M_2$ as a product of the form $M_1' \times M_2'$, where the process M_1' was constructed by adding self-loops for symbols in $\Sigma_2 - \Sigma_1$, and similar for M_2'.

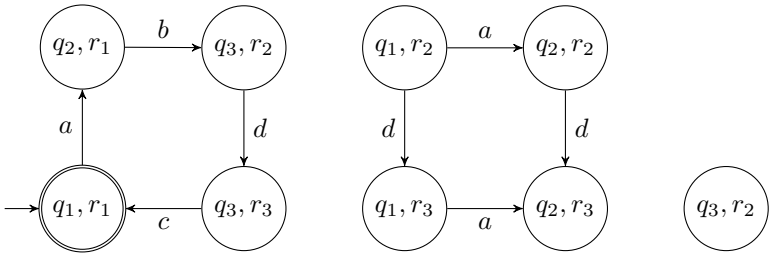

Figure 4: $M_{abc} \parallel M_{bdc}$.

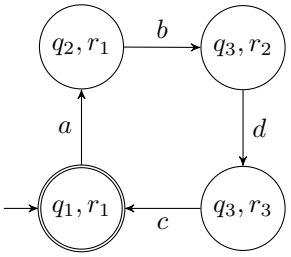

Figure 5: The minimised version of $M_{abc} \parallel M_{bdc}$.