Design Patterns for DFAs (Deterministic Finite Automata)

Agathe Merceron
TFH Berlin
merceron@tfh-berlin.de

DP 1 - Overview

Aim: Design a DFA (*Deterministic Finite Automaton*).

Problem: You want to design a DFA. The language the DFA should accept is arbitrary. It may be elementary. The language "all words over the alphabet {0, 1} that begin with 0" is an example of an elementary language. The specification of the language may be more intricate and contains words such as *or, and, not, from right to left* and so on. The language "all words over the alphabet {0, 1} that begin 0 and end with 0" is an example of an intricate specification that contains *and*. An intricate specification may also contain operators such as union, intersection, concatenation and so on. The language {w in {0, 1}* | w begins with 0} ∪ {w in {0, 1}* | w ends with 0} is an example of a specification that contains the union operator. What are the steps you should follow to obtain a (correct) automaton?

Solution: First look at examples. They do give good ideas and you may find your problem already solved. If you do not find the automaton you are looking for, then read the specification carefully.

If the specification is elementary, then try the pattern **3 Steps-Method**.

If the specification contains the word *or* or uses the union operator, then separate the language in two parts or sub-languages. Design a DFA for each part (if the specification is intricate, you may have to use other patterns while designing the DFA for a sub-language) and compose the two automata as explained in the pattern **Or**.

If the specification contains the word *and* or uses the intersection operator, then separate the language in two parts or sub-languages. Design a DFA for each part (if the specification is intricate, you may have to use other patterns while designing the DFA for a sub-language) and compose the two automata as explained in the pattern **And**.

If the specification contains the word *not* or uses the complement operator, then consider the positive language without the *not* and design a DFA for it (if its specification is intricate, you may have to use other patterns to design the DFA for it). With the pattern **Complementation** you get the desired automaton.

If the specification contains the words *followed by* or uses the concatenation operator, then separate the language in two parts or sub-languages. Design a DFA for each part (if the specification is intricate, you may have to use other patterns while designing the DFA for a sub-language) and compose the two automata as explained in the pattern **Concatenation**.

If the specification contains the words *from right to left* or uses the reverse operator, consider the language when inputs are read normally that means from left to right and design an automaton for it (if its specification is intricate, you may have to use other patterns to design the DFA for it). With the pattern **Reverse** you get the desired automaton.

DP 1.1 - 3 Steps-Method

Aim: Design a DFA from scratch.

Problem: You have to design a deterministic finite automaton. The language the automaton should accept is elementary. Examples of elementary languages are "all words that begin with 0" or "all words that contain an even number of 0's" (languages over the alphabet {0, 1}). Which steps should you follow to get a (hopefully) correct automaton?

Solution: First look at examples. They do give good ideas. If you do not find the automaton you are looking for and you do not feel sure how you should proceed, the 3 steps-Method may help.

First step: Design Tests. You should make sure, you understand the language the automaton has to accept. You should have examples of words that belong to the specified language and examples of words that do NOT belong to it. You will use these words later to test your attempt. Therefore the first step is to look for test-words, systematically ordering the words by length. We suppose that the alphabet is {0, 1}.
Words of length 0:
Does ε, the empty word, belong to the accepted language?
Words of length 1:
Do 0 and 1 belong to the accepted language?
Words of length 2:
Do 00, 01, 10 and 11 belong to the accepted language?
Words of length 3:
Do 000, 001, 010, 011, 100, 101, 110 and 111 belong to the accepted language? And so on till you have a fairly clear idea of the words that make up the language your automaton has to accept. At the end you come up with two lists ordered by length: a positive list that contains words the automaton should accept and a negative list that contains words the automaton should not accept.

Let us apply this step for the language L1 over the alphabet {0, 1}: L1={ w | w contains exactly one 0}.
Does ε, the empty word, belong to L1? No because ε is empty and therefore contains no 0.
Does 0 belong to L1? Yes.
Does 1 belong to L1? No.
Does 00 belong to L1? No because 00 contains two zeros and so on.

The result of this step looks as follows.
Accepted: 0, 01, 10, 011, 101, 110, 0111, 1011, 1101, 1110, 01111, ...
Not accepted: ε, 1, 00, 11, 000, 001, 010, 100, 111, 0000, ...
In that example the two lists contain infinitely many words. This is quite often the case.

Second Step: Design the Automaton. The purpose of a state of a finite automaton is to remember something about the word that has been read so far. So each state has a meaning or an interpretation that can be explained with a comment. This kind of comments in automata is similar to implementation comments in programs: They help the author to be convinced that the automaton does what it should, and they help others to understand how the automaton works. Read carefully the specification of the language and ask yourself, what is important and has to be remembered in states. This allows to construct states and transitions. Here also a systematic approach is useful. We suppose again that {0, 1} is our alphabet.
Words of length 0 - First state. Every Automaton has at least one state, the initial state. Let us call the initial state q0. An automaton is in that state at the beginning, which is to say after having read

ε the empty word. Ask yourself: what should q0 remember? What is ε for the accepted language? This leads towards a preliminary interpretation of this state. In case ε is accepted, q0 has to be an accepting state.

Let us go ahead with the language L1. An appropriate interpretation for q0 so far is "no symbol read and thus no 0 read". q0 is not an accepting state.

Words of length 1 - Second (and third) state. Except for the two trivial DFAs, that accept the empty language and the language composed of all words, DFAs have more than one state. To add further states and transitions you have to examine words of increasing length.

Let us begin with the word 0. What is it for the accepted language? Is the relevance of ε and 0 the same? If yes, it is not necessary to add a second state yet. You may have to adapt the interpretation of the first state and you add a transition from q0 to itself with symbol 0. If the relevance of 0 is completely different from the one of ε , you need to add a second state. This is in particular the case if ε is accepted but not 0, or the contrary. We call this second state q1. You write a comment for it and add a transition from q0 to q1 with the symbol 0. You proceed similarly considering ε and 1 as well as 0 and 1. In the case where the relevance of ε, 0 and 1 differ completely you end up with three states at that stage. In any case you have added two transitions, one for each input symbol. The four possibilities that can come up after having examined words up to length 1 with the alphabet {0, 1} are shown in Figure 1. (What are these possibilities with the alphabet {a, b, c} instead of {0, 1}?)

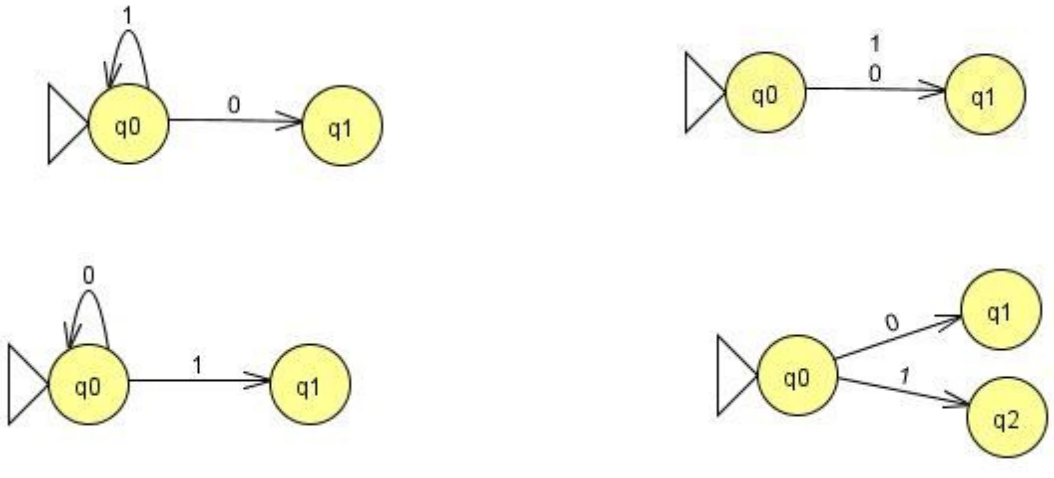


Figure 1. The four possibilities after having examined ε, 0 and 1.

To illustrate this step so far, let us come back to the language L1. ε and 0 cannot be put together simply because 0 is accepted but not ε. So we add a second state q1. At that point an appropriate interpretation for q1 is "First 0 read". q1 is an accepting state because 0 is accepted.
The relevance of ε and 1 for L1 is the same: "no 0 read yet". We adopt this interpretation for q0 (adaptation from the previous interpretation). That way we get the beginning of the automaton we are looking for as shown in Figure 2.

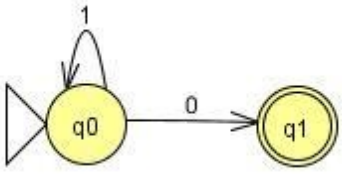


Figure 2. Beginning of the Automaton for L1.

Words of length 2, 3, ... - More states. To add more states and transitions you have to ask yourself what is the relevance of longer words such as 00, 01, 10 and 11, 000 and so on for the language? Does it differ from the one of the words you have already examined?

Let us begin with 00. First you have to realize that the second 0 is read after the first one. This means that for the second 0 you have to consider the state reached after having read the first 0. That state can be q0 or q1 (see Figure 1). For our running example L1, that state is q1. To stay general, we will call that state p0 (this denotation should be a reminder that one 0 has already been read).
You go ahead with the same kind of questions: What is 00 for the language? Is the relevance of ε and of 00 the same? If 'yes' possibly you adapt the comment of state p0 and you add a transition from p0 to the initial state for the input symbol 0 if it does not exist yet. Note that in case p0 is the initial state, the transition exists already. If 'no' examine 0 and 00, possibly 1 and 00 and proceed similarly. In case the relevance of 00 is different from the one of ε, 0 and of 1, add another state, say p00, write a comment for it and add a transition from p0 to p00 for the input symbol 0.

Proceed similarly with the word 01. After having examined its relevance with respect to ε, 0 and 1 do not forget to examine it also with respect to 00 before adding a new state.
Use the same approach for words of increasing length till you cannot add any new state any more.

Let us look again at L1 and the beginning of the automaton we have already got. The relevance of 00 for the language is not the same as the one of ε, 0 or 1, because two zeros have been read, and not exactly one as written the specification of L1. Therefore we add a third state that we call q2. "Two zeros have been read" is, at that point, an appropriate interpretation for q2.
In contrast, the meaning of 01 is the same as the one of 0: exactly one zero has been read. We adopt this wording as the new interpretation of q1 and add a transition from q1 to itself with the input symbol 1. 10 has also the same relevance as 0. The word 10 can be read with the automaton we have got so far. Therefore we do not need to add any state nor any transition. 11 has the same relevance as 1. Here again no need to add any state or any transition.

The words 000 and 001 have the same relevance as 00: "more than one 0 has been read". This wording becomes the interpretation of q2 and we add two transitions from q2 to itself for the input symbols 0 and 1. Words of length 3 or 4 do not bring any change to the automaton. The interpretation of states makes it clear that we have covered all what can happen while reading symbols with respect to the specification of L1. The complete automaton is shown in Figure 3.

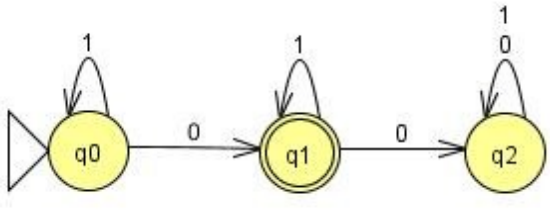


Figure 3. DFA for L1.

Interpretation of states:
q0 : no 0 read yet.
q1 : exactly one 0 read.
q2 : more than one 0 read.

Third step: Test. Check whether the automaton behaves the way it should with all words from your positive list and all words from your negative list. Be cautious to go through every single

transition and state while testing. You may have to design more tests to achieve this.

If your automaton passes your tests: congratulations. Most probably you have got everything right.
If testing does not work the way it should, double check two things: first, are your lists OK? That could be that you have put a word in the wrong list. If you feel sure of your lists, then check again the automaton and adjust it, till you are happy with the result.

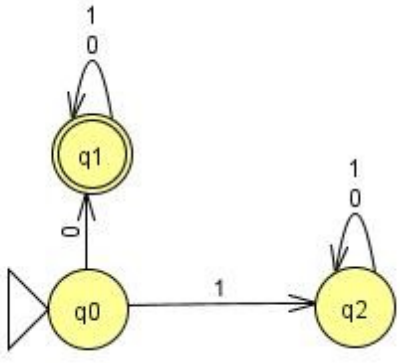
Discussion: A feature of this approach is to make you design tests before you design the automaton. In that respect there is a parallel with the software engineering approach "eXtreme Programming" (XP) [Beck 00].
DFAs are finite as their name says but they can accept infinitely many words. A DFA is a formal finite way of representing an infinite language.
Because DFAs are finite they have a finite set of states, and therefore what can be remembered about words that have been read is necessarily limited. Languages accepted by DFAs are called regular languages. Not every language is regular. In other words there are languages that can not be accepted or recognised by deterministic finite automata. It is not always obvious to anticipate or foresee whether a language is regular. As an example the language {w | w contains as many subwords 01 as subwords 10}is regular (would you like to design a DFA for it? Tipp: you do not have to remember how many subwords 01 or 10 the automaton has read so far, you need only to notice when the symbols 0 and 1 alternate). On the contrary it is impossible to design a DFA for the language {w | w contains as many 0 as 1}.
The latter is an example of a language which is not regular but context-free.

Exercises: Use this pattern to design DFAs for the following languages over the Alphabet {0, 1}. You can compare you answer with the one given here.
L2 = { w | w begins with 0}.
L3 = { w | w ends with 0}.
L4 = { w | w contains a positive even number of 0}.
L5 = { w | w is multiple of 3 when interpreted as a binary integer}.

Solutions to exercises

L2 = { w | w begins with 0}.

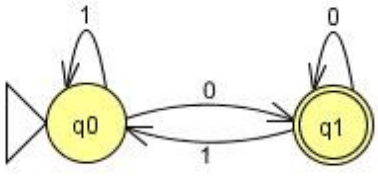
Accepted: 0, 00, 01, 000, ...
Not accepted: ε, 1, 10, 11, 100, ...
(The language is infinite because the positive list contains infinitely many words.)



Interpretation of states:
q0 : no symbol read yet.
q1 : the first symbol is not 0.
q2 : the first symbol is 0.

L3 = { w | w ends with 0}.

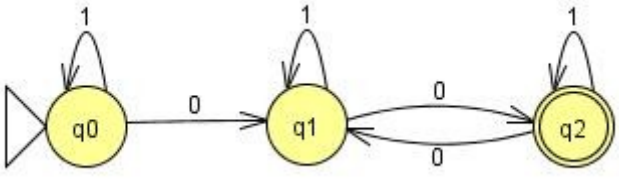
Accepted: 0, 00, 10, 000, 010, 100, 110, 0000, ...
Not accepted: ε, 1, 01, 11, 001, 011, 101, 111, 0001, ...



Interpretation of states:
q0 : no symbol read yet or the last symbol that has been read is not 0.
q1 : the last symbol that has been read is 0.

L4 = { w | w has a positive even number of 0}.

Accepted: 00, 001, 010, 100, 0000, ...
Not accepted: ε, 0, 1, 01, 10, 11, 000, 011, 101, 110, 111, 0001, ...

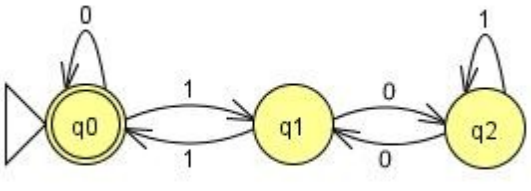


Interpretation of states:
q0 : no 0 has been read yet.
q1 : an odd number of 0 has been read.
q2 : an even and strictly positive number of 0 has been read.

L5 = { w | w ist a multiple of 3 when interpreted as a binary integer}.

Here we assume that ε, 0, 00 and so on are trivially multiple of 3.
Accepted: ε, 0, 00, 11 (integer 3), 000, 011 (integer 3), 110 (integer 6), 0000, 1001 (integer 9), 1100 (integer 12), 1110 (integer 15), … .
Not accepted: 1 (integer 1), 01 (integer 1), 10 (integer 2), 001 (integer 1), 010 (integer 2), 100 (integer 4), 101 (integer 5), 111 (integer 7), … .

Before you begin to design the automaton, it may be useful to remember a few things about binary numbers. The word 1 is also the integer 1 in binary. If 0 follows 1, we get the word 10, which is the integer 2 in binary and also 2 times 1. If 1 follows 1, we get the word 11, which is the integer 3 in binary and also 2 times 1 plus 1. Generally: reading a 0 multiplies the integer read so far by 2 while reading a 1 multiplies the integer read so far by 2 and then adds 1 (see the two lists above).
Checking the remainder of the number read so far when divided by 3 is a good trick to know whether the number read so far is a multiple of 3 or not. The remainder can be 0, 1 or 2. The value 0 says that the number read so far is a multiple of 3.



Interpretation of states:
q0: the remainder of the number when divided by 3 is 0.
q1: the remainder of the number when divided by 3 is 1.
q2: the remainder of the number when divided by 3 is 2 . See also exercise 2.2.6 (a) in [Hop 01].

DP 1.2 - Or

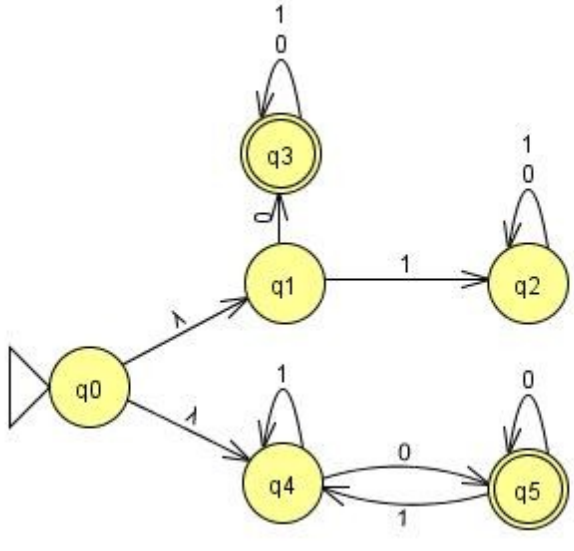
Aim: Design a DFA through the union of two other DFAs.

Problem: You want to design a DFA. The specification contains the word *or* or the language that should be accepted is defined as the union of two regular languages. An example is "all words over the alphabet {0, 1} that either begin with 0 or end with 0".

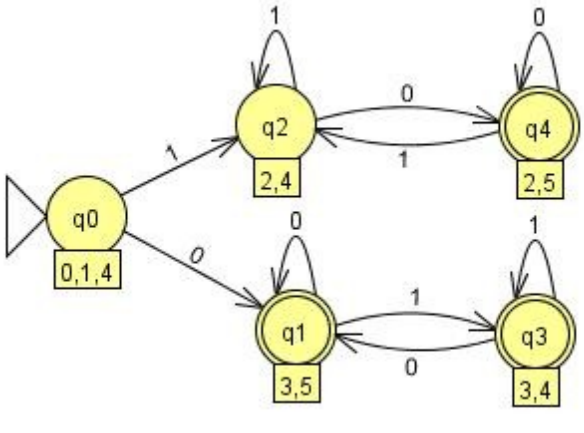
Solution: Design a DFA for each of the two sublanguages (if the specification is quite intricate, you may have to use other patterns while designing the DFA for a sublanguage). Compose the two automata you have got by adding an extra new initial state and an ε-transition from this new initial state to each of the old initial state. You have got a non deterministic finite automaton (NFA for short) with ε-transitions. This NFA accepts exactly the desired language. You just need now to transform this NFA into a DFA using the standard lazy evaluation approach [Hop. 01]. If needed, the DFA can also be minimised [Hop 01].

Example: Let L = "all words over the alphabet {0, 1} that either begin with 0 or end with 0". L is the union of two sublanguages L1 = "all words over the alphabet {0, 1}that begin with with 0" and L2 = "all words over the alphabet {0, 1}that end with with 0".

You can find DFAs for L1 and L2 in DP1.1. The or-composition of these two automata gives the following NFA with ε-transitions:



Its transformation into a DFA (done with [JFLAP 06]) gives the following DFA:



DP 1.3 - Complementation

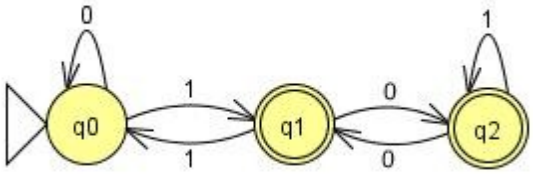
Aim: To design a DFA through the Complementation of another DFA.

Problem: You want to design a DFA. The language that the automaton should accept is defined as the complement of another regular language or its specification contains words such as *not, none* or *no*. An example is { w | w is *not* a multiple of 3 when interpreted as a binary integer}.

Solution: Design an automaton for the positive language (if the specification of the positive is quite intricate, you may have to use other patterns while designing a DFA for it). Obtain the complement automaton by exchanging accepting states and non accepting states. The automaton you get that way accepts exactly the words that the automaton for the positive language does not accept.

Example: Let L = { w | w is *not* a multiple of 3 when interpreted as a binary integer}. The positive language is L1 = { w | w is a multiple of 3 when interpreted as a binary integer}. A DFA for L1 is given in DP1.1.

The complement pattern gives the following DFA:



DP 1.4 - And

Aim: To design a DFA that accepts the intersection of two regular languages.

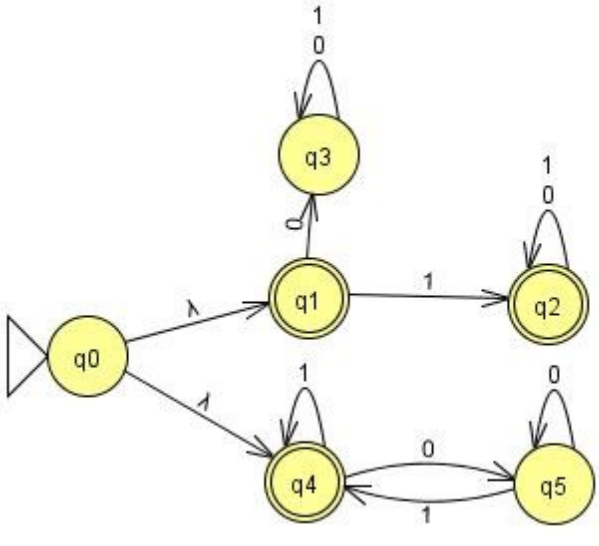
Problem: You want to design a DFA. The language it should accept is defined as the intersection of two regular languages or its specification contains the word *and*. An example is "all words over the alphabet {0, 1} that begin with 0 and end with mit 0".

Solution: The present solution is based on De Morgan's law. Let L and L' be two sets. One of the De Morgan's laws says:
L ∩ L' = complement (complement(L) ∪ complement(L')),
where ∩ denotes intersection and ∪ denotes union.

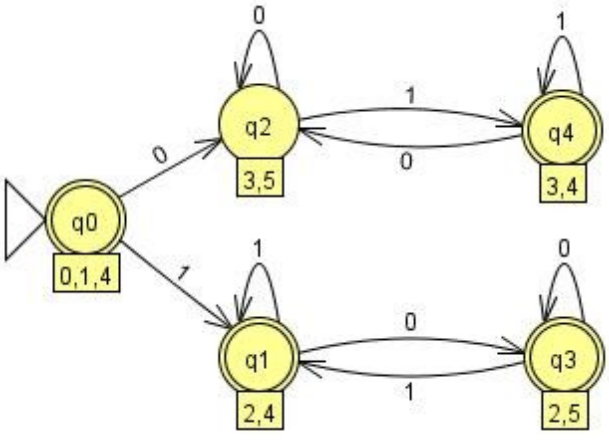
Using automata, you have to do the following. First you design a DFA for each sublanguage L1 and L2 (if the specification is quite intricate, you may have to use other patterns while designing the DFA for a sublanguage). Let A1 and A2 be the two DFAs that you have got. You use the pattern **complement** on A1 and on A2. You get two DFAs that we call CA1 and CA2. Now you compose with the pattern **or.** Transform the NFA you get from this pattern into a DFA that we call A3. Use again the pattern complement on A3. This gives you the DFA for the intersection. You may want to minimize it [Hop 01].

Example: Let L = "all words over the alphabet {0, 1} that begin with 0 and end with mit 0". L is the intersection of two sublanguages L1 = "all words over the alphabet {0, 1} that begin with 0" and L2 = "all words over the alphabet {0, 1} that end with mit 0".

You find DFAs for L1 and L2 in DP1.1. After applying the pattern **complement** on the DFA for L1 and on the DFA for L2 and then applying patter **or**, you get the following NFA with ε-transitions:



The transformation into a DFA (here done with [JFLAP 06]) gives the following DFA:



Now you only need the use pattern complement (that means exchange accepting states and non accepting states) to get the DFA that you are looking for. In the present example the resulting DFA can be minimised (q1 and q3 are equivalent).

Discussion: Another solution consists in building the product of two DFAs, see [Hop 01] S. 136.

DP 1.4 - Reverse

Aim: To design a DFA that is the reverse of another DFA.

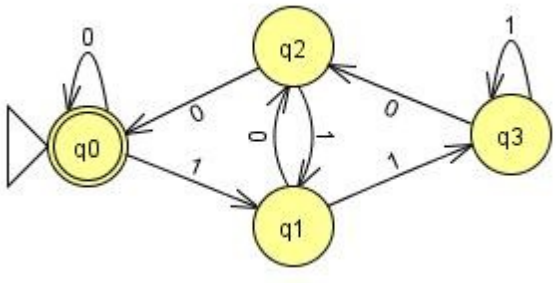
Problem: You want to design a DFA. The language that the DFA should accept uses the operator reverse or its specification contains words like *from right to left*. An example is { w | w is a multiple of 4 when interpreted as a binary integer and is read form right to left}.

Solution: Consider the language when input words are read normally, that means from left to right and design a DFA for it (if its specification is quite intricate, you may have to use other patterns while designing the DFA). Obtain from A a NFA with ε-transitions doing the following:

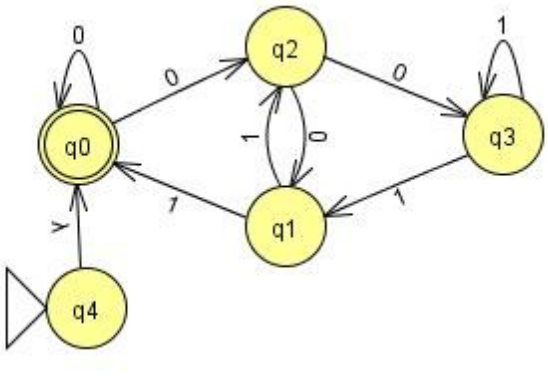
1. reverse all arcs of the transition function,
2. make the initial state of A the sole accepting state of the NFA,
3. and add a new initial state p and ε-transitions from p to all former accepting states.

Example: **Let** L = { w | w is a multiple of 4 when interpreted as a binary integer and is read form right to left}.

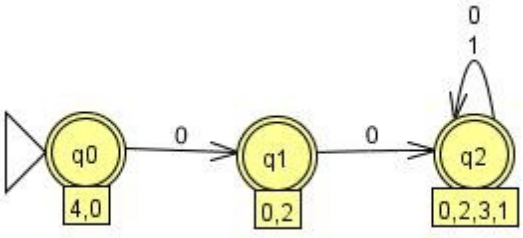
Wenn input words are read normally, using an approach like the one in exercise 5 DP1.1 you get the DFA below:



The reverse pattern gives you the following NFA with ε-transitions:



The transformation into a DFA (here done with [JFLAP 06]) gives the following DFA:



References

[Beck 99] Kent Beck (1999): Extreme Programming explained: embrace changes, Addison-Wesley Professional; second edition 2004 with Cynthia Andres.
[Hop 01] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2001): Introduction to Automata Theory, Languages, and Computation, Pearson Addison-Wesley.
[JFLAP 06] JFLAP Version 6.0 http://www.jflap.org/