STI 4A
Tools for Program Proof and Formal Verification

Work in progress. Do not distribute outside of INSA CVL.

Vincent Hugot — `vincent.hugot@insa-cvl.fr` — SA 2.24

March 29, 2024

Part I

Lab classes

1 Preliminaries (preferably before the first lab class)

This class is supported by a Python implementation of Nondeterministic Finite State automata, which I provide. You will need a machine correctly configured for Python development.

1.1 Setting up a work environment

You may use the INSA's machines or your own, as you prefer; whatever works best for you. The same goes for your choice of Python editor. I provide recommendations based on what I have used and tested.

1.1.1 Operating System

Linux is *strongly* recommended, though I have tested the code under Windows. Once.

Some students have had success running things on the **Windows Subsystem for Linux** (WSL), which is, from my understanding, functionally equivalent to running a Linux VM on Windows, but probably more efficient, as it seems to rely on compatibility layer rather than full virtualisation. Some other students have reported issues such as RAM being gobbled up during downloads (!?), slowing the system to a crawl. Your mileage may vary.

1.1.2 Choice of Linux distribution

Most Linux distributions will do, but some are easier than others to set up. **Debian-based distribs** (*Ubuntu, Mint,. . .) tend to ship with old software, with only security updates provided, which is a problem as we need a very recent version of Python, in particular. We usually end up compiling Python from source, or using PPA.

Some versions of them, starting from 2018, also lack the `pdftk` package, because of a packaging bug. I have observed that the overall experience of setting up a Debian-based work environment has been painful for many students.

Arch-based distributions, on the other hand, ship with up-to-date software, and I have found them much more straightforward to set up. Python may be two or three month behind the latest, at most.

The choice is yours.

1.1.2.1 Provided VM: Arch-based system

***TECHNICAL ISSUES:** If VMware fails with `The import failed because .ova did not pass the OVF specification conformance or virtual hardware compliance checks`, click "Retry with lower specifications".*

For your convenience, I have prepared an **Arch Linux + KDE VM**, following the instructions detailed in the next paragraph, *which you should read anyway, to understand what's inside it*.

`https://files.vhugot.com/Restricted/Verif/VM/`

Do not forget to update `NFA_Framework` with `git pull` at the beginning of each class.

The `sudo` password is `aaa`, same as the main user's name.

1.1.2.2 Instructions: Arch-based system

If you already have an Arch set up, through whatever means, use that.

If not, to get things running in reasonable time, I strongly recommended using EndeavourOS (`https://endeavouros.com/`), which is basically a nice installer for Arch. That is what I use on all my machines. [a]

EndeavourOS ships with KDE by default on the ISO, which I recommend.

Once the system is set up, download run the install script, and you should be good to go:

`https://github.com/vincent-hugot/NFA_Framework/blob/main/NFA_install_scripts/arch_endeavouros.sh`

Very optional: LaTeX output The installation of LaTeX, which the script does not perform by default, requires several GB and is not really needed for this course. You can un-comment the corresponding line in the script to install everything.

During `./tests.py`, you will see messages of the form

```
pdflatex is not installed: aborting LaTeX content (normal for students)
```

This is not a problem. I use the LaTeX output to generate the nice automata sagittal diagrams in this document; you are not writing lecture notes so you probably don't need that feature.

Arch-Linux package management in 30s

You can then install your preferred editor using `pacman -S <foo>`. A package list is on `https://archlinux.org/packages/`. For instance, `pycharm-community-edition`.

There is also `https://aur.archlinux.org/packages/` for user-provided packages. Those can be installed via `yay -S <foo>`. For instance, `visual-studio-code-bin` is on the AUR.

[a] ArcoLinux is another possibility. Then there is the `archinstall` script from base Arch. (Manjaro is also well-known and Arch-based, but there are a few complications, with more distro-specific repositories, packages being held back etc.)

1.1.2.3 Instructions: Debian-based system

Use the script:

`https://github.com/vincent-hugot/NFA_Framework/blob/main/NFA_install_scripts/debian_ubuntu.sh`

Tested on a Kubuntu 23.10.

`python3-pylsp` may be absent from older distributions; it is not essential and can be removed.

On some versions of Debian/Ubuntu, the package `pdftk` is absent. `sudo snap install pdftk` should work (snap is an alternative, somewhat universal, package installer). Of course, `snap` itself may not be installed by default. . .

If the packaged version of lark is too old, `pip install lark` should do the trick; remove `python3-lark` before doing that, of course.

The same remark applies regarding LaTeX as for Arch-based systems.

1.1.2.4 Instructions: Fedora / SUSE-based system

Use the script (kindly provided by a student):

`https://github.com/vincent-hugot/NFA_Framework/blob/main/NFA_install_scripts/fedora_suse.sh`

The same remark applies regarding LaTeX as for Arch-based systems.

1.1.2.5 Instructions: Microsoft's Spyware OS

I do not recommend you use Windows for this. Or anything at all, really. Ever.

Tested on version 10.

(1) Wait for the OS to reboot three times in a row for updates that must be more cosmically important than you getting any work done.

Tell the OS you don't want Edge, don't want to link with a Microsoft account, don't want to pay for a "premium" cloud service, don't want to be nagged about this ever again. . . *wait*. You can't tell it "no", just "remind me in three days". My mistake, I thought you were your computer's boss for a minute, how silly of me.

Oh, fun fact, when you search for "NFA_Framework" on Bing (Microsoft owned), the first few links are for Microsoft's .NET Framework. How quaint.

(2) If needed, install Git: `https://git-scm.com/download/win`. The "Git Bash" is nice.

(3) If needed, install Python, adding it to the system's `PATH`.

(4) Install `dot`: `https://graphviz.org/download/`. Jump through three flaming hoops to convince Edge that just because this is not "commonly downloaded" you know what you're doing and really, *really* want to keep it.

Then jump through two more flaming hoops for Windows Defender, arguing that *yes*, you'd really like to run it, though it is "unrecognised". Anything that not everybody else is doing is bad, don't you know?

Do not forget to add it to the system's `PATH` when the installer asks.

(5) Install `pdftk` `https://framalibre.org/content/pdftk`, putting it in the system's `PATH`.

(6) Additionally, `pip install more_itertools lark`

(7) At last,

```
git clone https://github.com/vincent-hugot/NFA_Framework.git
```

and you're golden. Well, you're still using Windows, so "golden" is perhaps a bit strong. You're *okay*.

Note that the default fonts on Windows lack support for some — or, it seems, *all* — Unicode math symbols, so characters will be missing from the pdf renders, in particular the titles of some automata.

That's a major problem for CTL formulæ (an important part of this course) which are basically made entirely of fancy-schmancy symbols.

The only font I found with the right symbols is Cambria Math, but it has other issues that render it unsuitable.

(8) To have more symbols, install the fonts `https://github.com/alerque/libertinus`.

For some reason some symbols are still missing from the Sans variations, though they are fine *for the very same font* under Linux. But at least the CTL formuæ are readable — the "Until" symbol is missing, though.

Thus, if Libertinus Math is installed, but not Libertinus Sans, the system will choose it. Some product symbols are still missing, but the CTL formulæ should be fully legible.

You can also manipulate the font directly through setting `NFA.VISUFONT`.

(9) If you try and generate the pdf while it's open in a PDF reader like Foxit or Adobe, you will get

```
PermissionError: [WinError 32] The process cannot access the file
because it is being used by another process: 'visu.pdf'
```

because Windows does not handle file access like Linux. You can't just have the reader detect changes and refresh the view automatically.

Browsers do not keep the file open after loading, so you can use that and refresh the "page" with F5.

(10) Some things may be wonky in the cmd/powershell, like spinners/progress bars etc that I may not have written in a cross-platform way. If that bothers you, submit a patch.

1.1.3 Check that it works, and brush up on stuff

Now that you have things running on whichever OS you chose, there remains to check that the test output makes sense, and get going.

(1) Check that everything works correctly by running `wolf.py`, `tests.py`, and `lecture_automata_products.py`. Compare the output of the latter two to the PDFS provided on Celene. (The install scripts automatically run `tests.py`).

Note: There could be some variation between your output and the PDFS *because* **(1)** *the exact appearance of graphs – by which I mean node placement – can vary "randomly" from one execution to the next, and* **(2)** *I have probably altered the sources since I uploaded the* PDFS *— I won't update those systematically. The idea is to check whether you get a crash or something legitimate-looking, not to ensure the output is the same pixel-for-pixel.*

(2) Brush up on Python a bit. For instance, if you didn't "get" comprehension expressions last year, reading the relevant section of the lecture notes would be helpful: my code makes heavy use of them.

(3) Likewise, brush up on finite automata theory. If you don't remember what $\langle \Sigma, Q, I, F, \Delta \rangle$ stand for, you'll be a little bit lost.

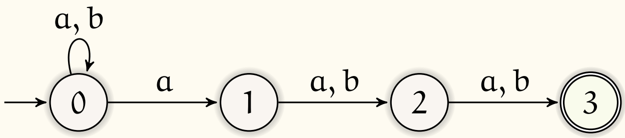
2 Basic finite state systems

Let's brush up on automata and familiarise ourselves with the NFA framework.

The automata framework is in `nfa.py`, which depends on `toolkit.py`. All other files depend on `nfa`. There are many examples in `lecture_automata_products.py` and `tests.py`.

Create a file `basic.py` and follow along with the examples in this section.

Recall the automaton seen last year (if you were here), recognising the words whose antepenultimate letter is a:



It has the following transition table:

		a	b
Initial	0	0, 1	0
	1	2	2
	2	3	3
Final	3		

Let's implement it in the framework. We have the constructor `NFA(I,F,Δ)`, which is pretty self-explanatory, with an optional argument `name`, generally just used for display purposes — a major exception being named synchronised products, which we shall see later in this class, where the automata's names are actually significant. Read the constructor's documentation and code for more information.

```
A = NFA( {0}, {3},
        { (0,'a',0), (0,'b',0), (0,'a',1),
          (1,'a',2), (1,'b',2),
          (2,'a',3), (2,'b',3) },
        name="a__")
```

If you print `A`, you get something like

```
NFA a__:   ## = 29
Σ 2  {'a', 'b'}
Q 4  {0, 1, 2, 3}
I 1  {0}
F 1  {3}
Δ 7  {(0, 'a', 0), (0, 'a', 1), (0, 'b', 0), (1, 'a', 2),
       (1, 'b', 2), (2, 'a', 3), (2, 'b', 3)}
```

Note that the states Q and symbols Σ are computed from the provided transitions.

You can use `A.visu()` to visualise `A` in auto-formatted PDF form. Note that `A.visu()` returns `A`, so you can write directly

```
A = NFA(...).visu()
```

on one line. Generally, most algorithms in the NFA framework return an automaton, so you can chain operations on one line.

When writing automata "in extenso" — that is to say, by writing every transition by hand, as opposed to generating them in a **for** loop or something of that sort — you might want to use shorthand notation, with the `NFA.spec` method:

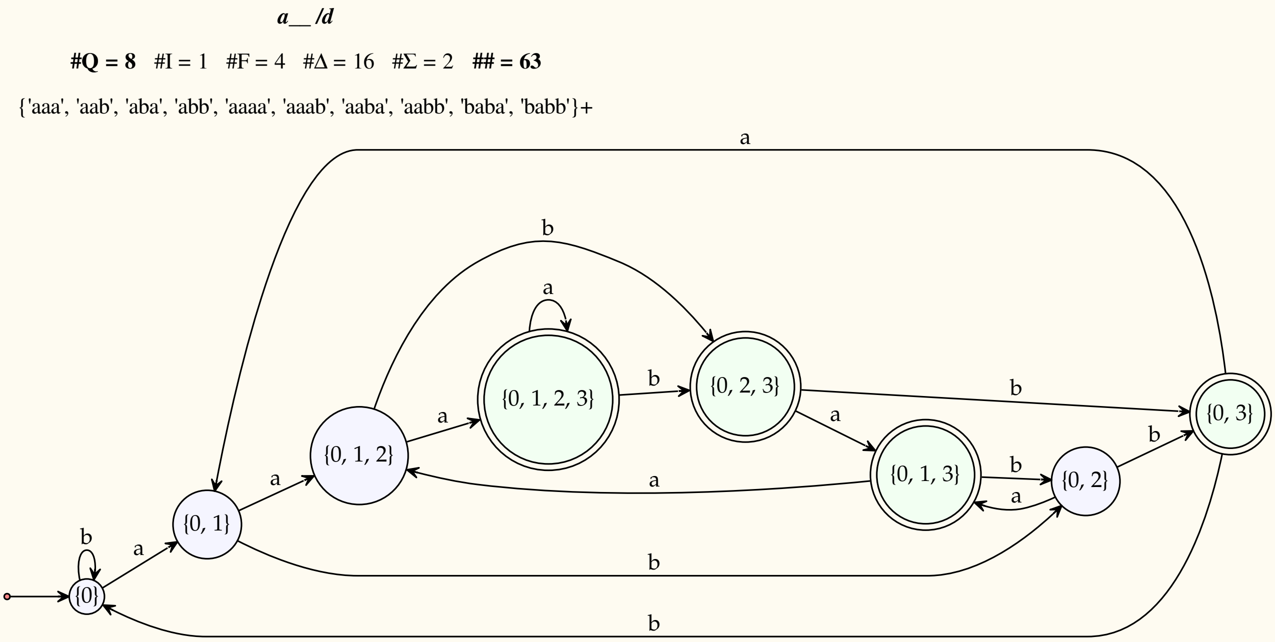
```
A = NFA.spec("""
    0
    3
    0 a 0 b 0 a 1
    1 a 2 b 2
    2 a 3 b 3
    """, name='a__, bis')
```

Read the doc / code, and observe the examples to infer how that syntax works.

Now let us make it deterministic: use

```
A.dfa().visu()
```

to visualise the determinised automaton. You will recognise the result of an exercise we did by hand last year:


a__/d
#Q = 8 #I = 1 #F = 4 #Δ = 16 #Σ = 2 ## = 63
{'aaa', 'aab', 'aba', 'abb', 'aaaa', 'aaab', 'aaba', 'aabb', 'baba', 'babb'}+
{0}
{0, 1}
{0, 1, 2}
{0, 1, 2, 3}
{0, 2, 3}
{0, 1, 3}
{0, 2}
{0, 3}
a
b

With the `NFA.table` method, you can print in the standard output, as a side effect, LaTeX code for a nice table of transitions.

If LaTeX is installed (which is not necessary for this course) you can display that table directly in the PDF by calling

```
A.dfa().visu_table()
```

This displays:

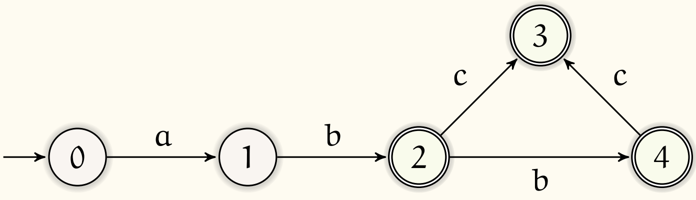
		a	b
Initial	{0}	{0, 1}	{0}
	{0, 1}	{0, 1, 2}	{0, 2}
	{0, 2}	{0, 1, 3}	{0, 3}
Final	{0, 3}	{0, 1}	{0}
	{0, 1, 2}	{0, 1, 2, 3}	{0, 2, 3}
Final	{0, 1, 3}	{0, 1, 2}	{0, 2}
Final	{0, 2, 3}	{0, 1, 3}	{0, 3}
Final	{0, 1, 2, 3}	{0, 1, 2, 3}	{0, 2, 3}

Automata naturally act as iterable containers for their recognised (possibly infinite) language. For instance, let us take

```
A = (NFA.of_set(["a","ab"]) + NFA.of_set(["b","bc"])).mini().renum()
```

Note: `NFA.of_set(S)` creates an automaton that recognises exactly the words in the (finite) iterable `S`, and the `+` operation on `NFA` is language concatenation. `NFA.mini` minimises the automaton (it takes care of ε-removal and determinisation if necessary), and `NFA.renum` renames the states into `0, 1, ...` by order of accessibility (by default).

We obtain:



`A` is iterable. In that case the language is finite, so we can simply write things like

```
>>> list(A)
['ab', 'abb', 'abc', 'abbc']

>>> for w in A: print(w,end=' ')
ab abb abc abbc
```

Words are generated from smallest to greatest length. Note that if the automaton contains ε-transitions, words may appear twice; in this case convert into `set` to avoid duplicates.

What if the language is infinite? `NFA` are slicable. The code below means "take up to the first two/ten elements of the language":

```
>>> list(A[:2])
['ab', 'abb']

>>> list(A[:10])
['ab', 'abb', 'abc', 'abbc']
```

This will always terminate, even if the language of `A` is infinite, whereas `list(A)[:2]` would not.

`len(A)` returns either the cardinality of the language, if it is finite, or `math.inf` if it is infinite.

Other operators and methods of interest on `NFA` include `|` for ∪, `&` for ∩, `+` for concatenation (compatible with strings and lists or sets of strings for adding finite languages as prefix or suffix), `*` (with `int`) for repeated self-concatenation, `-` for complement, `^` for symmetric difference (XOR for languages), `@` for language shuffle (we'll see what that is later), `.map` for states and transitions mapping / renaming, `.rm_eps` for epsilon removal, and `.trim` for trimming (removing all useless states, that is to say, states that are not on any path from initial to final states).

Trimming in particular is very useful; remember its existence. When modelling problems where you can lock yourself in a losing position without *immediately* realising it, you'll generate lots of "dead" states; trimming will remove them. The Indiana Jones problem, on which we'll spend some time, is an example of that.

You can also convert regular expressions into automata with the `renfa` module; for instance, the following code converts $(\varepsilon \mid ab)^*$ into a minimal DFA, showing most of the steps:

```
from renfa import E
( E("ab") | E("") ).star().show_all()
```

The line

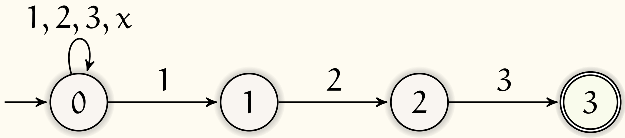
```
NFA.pdf_renderer.print_status()
```

is useful to put as the last line of any script in which there are lots of calls to `.visu`. Each such call is non-blocking and initialises a PDF rendering job. The PDF rendering jobs are collected and processed on every core available on the CPU. The line above gives you a progress indicator telling you how many rendering jobs are pending.

With this, you should be starting to get an idea of how to use the NFA framework. Play with `lecture_automata_products.py` and `tests.py` to go farther. Remember that you have access to all the code.

(1) Digicode: to warm up, implement the nondeterministic automaton for the "123" digicode seen in the lectures, and make it deterministic using the `NFA.dfa` method.

Recall that this automaton is of the form:



(2) Incrementable Integer Variable

Do it yourself first, even though the solution is in Sec. 8.4.4[p40]: "Incrementable Integer Variable".

The following was an exercise given in the 2019–2020 final exam, and will be very useful for several problems that involve counting.

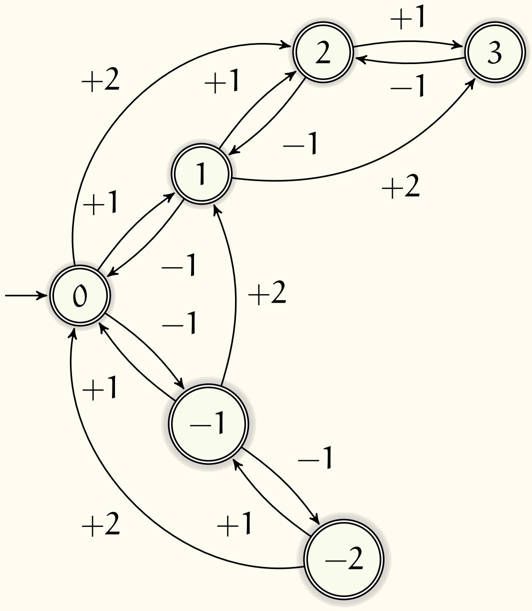
For all $n, m, i \in \mathbb{Z}$ and $X \subseteq \mathbb{Z}$, formally define a NFA $V(n, m, i, X)$ representing an integer variable on the interval $[\![n, m]\!]$, initialised to i, that can be incremented by the quantities in X, and only by those.

(We speak of *decrementation* when the quantity by which we increment happens to be negative.)

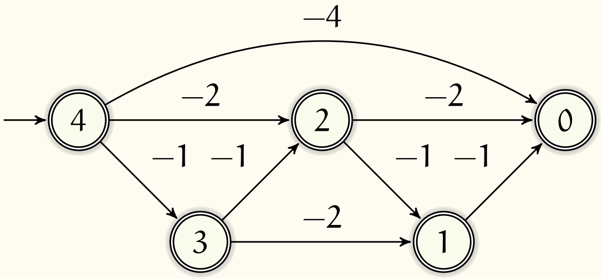
We permit neither overflow nor underflow. We consider all states final.

For instance, we have:

$V(-2, 3, 0, \{-1, 1, 2\}) \;=$



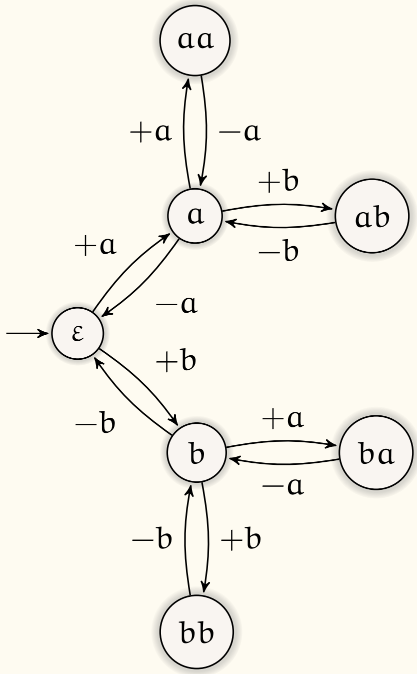
$V(0, 4, 4, \{-1, -2, -4\}) \;=$



(3) L'y faut le LIFO: For all $n, m \in \mathbb{N}^*$, model the finite-state behaviour of a LIFO (*Laboratoire d'Informatique Fondamentale d'Orléans*, errrr, I mean, **l**ast **i**n, **f**irst **o**ut) queue (or stack) on n distinct symbols *a*, *b*,. . . , of capacity m. See below for an example of LIFO$(2, 2)$, and further indications.

By *model*, in this and further questions, I mean: write the transition system as a function of the parameters n, m, implement it in Python, and visualise a fair number of instances to see how the parameters affect the systems.

$\mathrm{LIFO}(2,2) =$



Start with defining, mathematically, the automaton as a tuple $\langle \Sigma, Q, I, F, \Delta \rangle$, where each component may depend on m and n. See for instance Sec. 8.4.4$_{[p40]}$: “Incrementable Integer Variable”. Try to do that on your own, then you can check the solution given in the lecture notes: Sec. 10.0.1$_{[p43]}$: “FIFO / LIFO(n, m)”.

There are several ways to go about implementing this in Python. **You must implement each of them**, as they prepare for later exercises.

a. The first, and most “mathematical”, in the sense that it matches the mathematics very closely, with no additional algorithmics, is to compute the set of all states — here all strings of length at most m — and the rest proceeds as direct translation of the mathematics, preferably using set comprehension syntax.

Here is a proposal for the computation of the states (a variant of a question in 2020’s Python exam ;-)

```
def all_str(al, n):
    """Return the set of all strings of length at most n
    on alphabet al"""
    if n ==0: return {’’}
    return (rec :=all_str(al,n-1)) \
           | { w+c for w in rec if len(w) ==n-1 for c in al }
```

Thus the expected answer should be quite terse, and of the form

```
def lifo(n,m):
    symbs = ...
    states = all_str(symbs,m)
    return NFA(...)
```

b. Another way, which is conceptually interesting when you can't easily compute the set of reachable states in advance, — and we shall see such cases later — is to *grow* the automaton from its initial state, adding valid transitions, and therefore new states, then adding transitions for those new states, and so on, until we reach a fixed point where no new transition or state can be added.

Here is a skeleton for this approach:

```
A = NFA({""},{}, { }, name=f"LIFO({n} symbs, {m} cap)" )

q = 0
while len(A.Q) > q:
    q = len(A.Q)
    for p in A.Q.copy():
        # use A.add_rule; will update A.Q automatically

return A
```

Note: the `.copy()` in `for p in A.Q.copy()` is necessary because Python — quite legitimately — dislikes having an iterable altered while it is being iterated upon.

c. The `NFA` framework offers a more systematic way to handle such growth: see the `.growtofixpoint` and `.try_rule` methods. With them, you write a growth procedure that returns whether it added anything new (Boolean), and `.growtofixpoint` will automatically iterate this procedure until a fixed point is reached.

`.try_rule` is like `.add_rules`, but returns whether the rule is actually new (Boolean, again), which is useful to write growth procedures.

The growth pattern becomes:

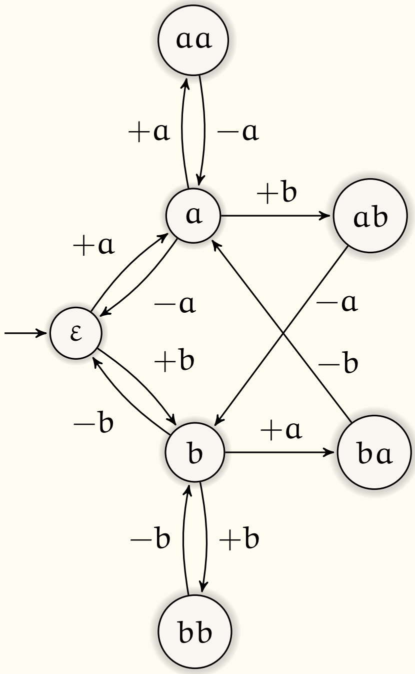
```
A = NFA({""},{}, { }, name=f"LIFO({n} symbs, {m} cap)" )

def grow(A):
    has = False # have I grown ?
    for p in A.Q.copy():
        # use "has = A.try_rule(....) or has" pattern
    return has

return A.growtofixpoint(grow)
```

An interesting functionality of `.growtofixpoint` is that, if you pass the optional argument `record_steps=True` to it, you can then use `.visusteps()` on the generated automaton to visualise, step by step, which states and transitions were added; which is pretty neat, if I do say so myself ;-)

At this point, you can move on to the exercices on complex systems (`wolf.py` etc); the other exercices on "basic" automata are optional.

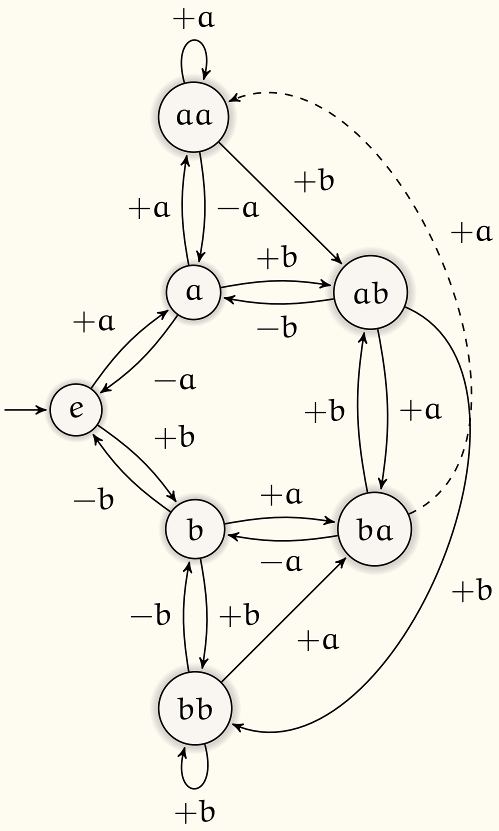
(4) Fee-fie-fo-fum, FIFO for fun: Similarly, for all $n, m \in \mathbb{N}^*$, model the finite-state behaviour of a FIFO (**f**irst **in**, **f**irst **o**ut) queue on n distinct symbols, of capacity m.

$\mathrm{FIFO}(2, 2) =$



(5) Forgetfulness: Now model the behaviour of LIFO and FIFO, under the same parameters, if the oldest stored elements are discarded when new elements are added while the stack/queue is already at capacity.

$\mathrm{LIFO}_f(2,2) =$



(6) Don't be a Dyck

A **Dyck word** is a correctly balanced word composed of [and]. By correctly balanced I mean that opening and closing brackets are paired correctly. A good definition is that by removing occurrences of consecutive pairs [], a Dyck word word can be reduced to ε. Here are a few Dyck words:

ε, [], [][], [[]], [[][]], [[]][], [][[]], [[[]]], [][][], [[[][]]], . . .

Here are a few words that are not Dyck words:

], [,][,]], [[,]]], []],][],][[, [[], [[[,]][, [][, []]],]]][,]][[,][[[, . . .

The **Dyck language** is the language of Dyck words.

Tip: we probably looked at Dyck words last year in automata theory, although that might depend on which group you were in.

a. Write a function $D(d)$ — mathematically, on paper, and then in Python as `Dyck(d)` — returning an automaton accepting the Dyck words of depth at most d.

By *depth*, I mean the level of imbrication of the brackets. For instance, [[]], [[]][], and [[]][[]] are of depth 2, and [][] is of depth 1.

If you have trouble understanding how you can define an automaton that depends on a parameter, recall that an NFA is a $\langle \Sigma, Q, I, F, \Delta \rangle$; here, each component may depend on d. See for instance Sec. 8.4.4[p40]: “Incrementable Integer Variable”.

b. Now, implement a function $N(d)$ that returns an automaton accepting the *complement* of $D(d)$. Is that the same thing as accepting all non-Dyck words?

Tip: My NFA framework implements the Boolean operators on automata. Find the right operator to use.

c. How many states would an NFA have to have to recognise the Dyck language?

d. Now, let us generalise this to Dyck words with several types of parentheses. They all must be correctly paired, for each type and between each type. For instance, here are a few Dyck words on pairs (), [], <> {}:

<> [[]], [(())], [][[]], < {} > [], {[] <>}, []{}{}, <> ({}), {}{}(), [{}()], ([]()), [[][]], ([]){}

However, note that ([)] is not a Dyck word, despite the fact that each type, considered in isolation, is balanced. They must be balanced with respect to each other.

Write a function `Dyck2(d,pairs)` returning an automaton accepting Dyck words recognisable with a stack of size d, with the different pairs specified as an even-length string. (I mean one stack total, not one stack per type of parentheses.) For instance, the words above are recognised by `Dyck2(3,"()[]{}<>")`[b].

Technical tip: You can pass `rankdir="TB"` (top-to-bottom) to `.visu` to change the orientation from the default left-to-right; useful for very wide trees, which this might become...

e. **Shuffled Dycks:** *(Best after Sec. 10.1.2[p51]: “Fully Unsynchronised Product ||: the Shuffle”)*

Now, let us remove the requirement that different types of parentheses must be balanced with respect to each other. Write a function `sDycks(d,pairs)`, with the same type of arguments as `Dyck2`, but recognising the words where the projection on each pair of parentheses — that is to say, if we ignore all other symbols — is a Dyck language for that pair. Thus we recognise all words of `Dyck2`, but also words such as w = ([)], which is not accepted by `Dyck2`, but whose projections $\pi_{()}(w) = ()$ and $\pi_{[]}(w) = []$ are Dyck words.

For instance, `D = sDycks(2,"()[]{}")` accepts:

ε, [], (), {}, ([)], [(]), {()}, ({)}, (){{}}, {[[}]], [{]}(), [(][]), [{}](), [{}()], ([)]{}, [{(]}), ...

Tip: You can use a shuffle product (||, `@` in the `NFA` class) on the relevant `Dyck` automata. You could also use several independent stacks or counters.

[b] Little bug of syntactic colouring on braces here... Ignore it.

You should find 27 states in `D`; furthermore, it is hard to read, with 108 transitions... Let's verify that it does what we expect.

Project `D` upon a given type of parentheses, for instance []. This can be done with `NFA.proj`. Since projection can balloon the number of transitions (here it goes to $|\Delta| = 324!$), minimise the result before visualising it:

```
D.proj("[]").mini().visu()
```

You should obtain an automaton equivalent to `Dyck(d)`.

Now, project on mismatched parentheses:

```
D.proj("(]").mini().visu()
```

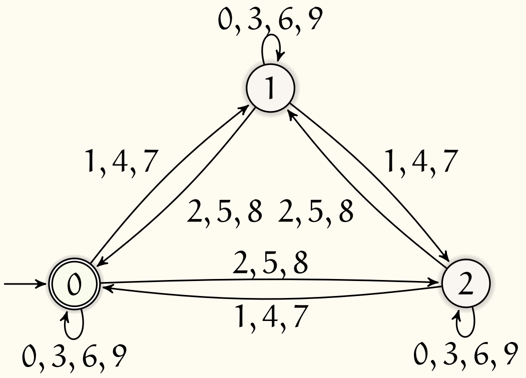
You should obtain a universal automaton.

(7) Modulos

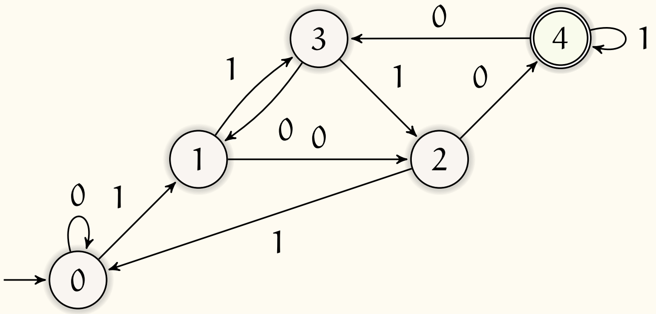
Define and implement automata $\text{modulo}(n, m, b)$ that have as language the string representations of base b natural integers that are congruent to m modulo n.

For instance, we have

$\text{modulo}(3, 0, 10) =$



$\text{modulo}(5, 4, 2) =$



I need not remind you that a simpler version of this exercise was done last year in the automata theory class, because of course you were paying rapt attention...

Using the modulo(n, m, b) function and automata minimisation, find a simple criterion for divisibility by 5 in base 10. Same question for for divisibility by 3 in base 6.

(8) The debatable elegance of tennis scoring

Here is an informal description of the scoring system of a tennis game, taken and adapted from Wikipedia:

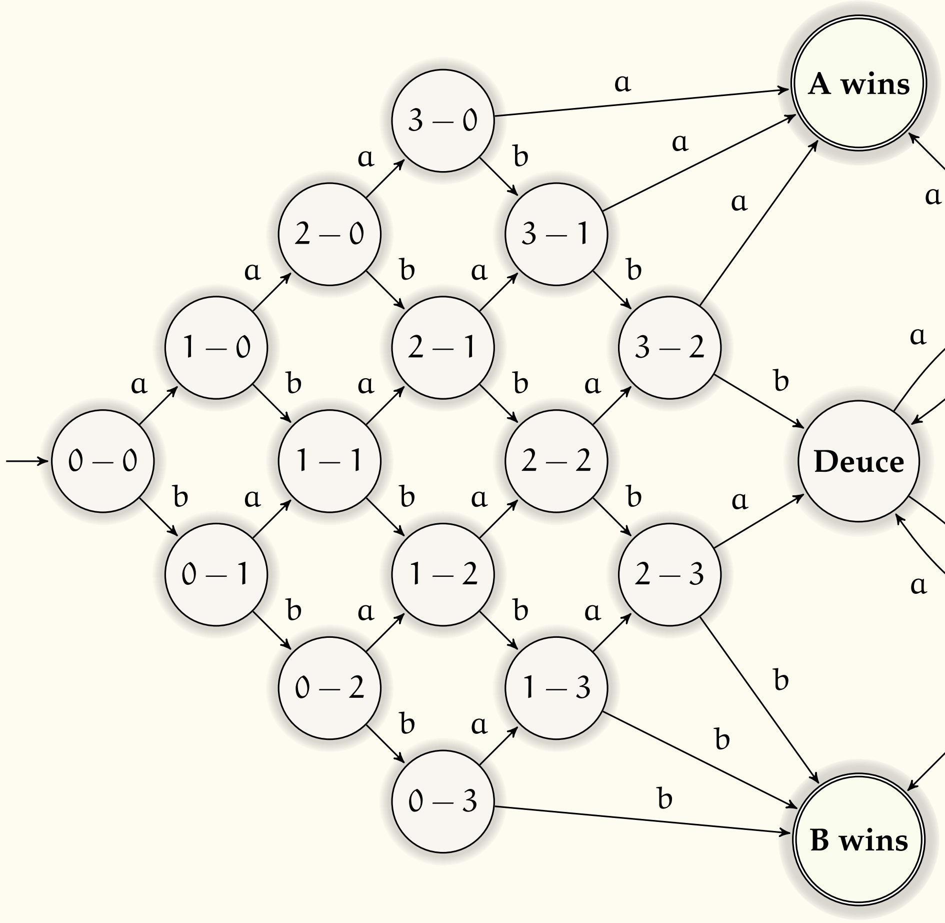
> *A game consists of a sequence of points played with the same player serving. A game is won by the first player to have won at least four points in total and at least two points more than the opponent.*
>
> *[. . .]*
>
> *If at least three points have been scored by each player, making the player's scores equal at 3 apiece, the score is not called out as "3–3", but rather as "deuce".*
>
> *If at least three points have been scored by each side and a player has one more point than his opponent, the score of the game is "advantage" for the player in the lead.*

Implement an automaton representing the evolution of a tennis game. The symbols a and b will be used to denote a point won by players A and B, respectively. The automaton will accept all sequences of points leading to a victory by either player. The states of the automaton shall have meaningful names.

Of course, you shall do most if not all of this by programming the logic of the game, not by writing the automaton in-extenso.

Tip: you may want to generate more states than there are, and then collapse and rename them through uses of the `NFA.map` method.

You should obtain something like that:



This is another exercise done last year — by hand, and probably only schematically, as there are 20 states.

3 Modelling complex systems using products

Trigger warning: lots of rivers, bridges, and boats in this section.

(9) wolf.py: Solution of problem seen in lectures.

Have fun with it, get a sense of how it works; you will need to apply what you see in there, along with the content of the lectures, to solve other problems.

(10) Indiana Jones and the Temple of Verification:

Indiana Jones, his annoying girlfriend, a wounded guy, and a whiny kid find themselves in a dire predicament: savage cannibalistic cultists are on their heels; in 15 minutes, they will be toast. . . or *on* toast.

Their only hope? swiftly crossing the crocodile-filled ravine, using the threadbare, rickety bridge. It is quite clear that the bridge can only support the weight of two persons at most — even if one of them is a kid.

To make things worse, night has fallen, and the bridge is far too treacherous to walk blind; a torch is necessary to examine the worm-eaten planks *before* setting foot on them.

Dr. Jones, being a seasoned adventurer, does have a torch in his inventory; the group will have to find a way to share. Though nobody else has a torch of their own, all can use Dr. Jones' torch to cross the bridge on their own or in pairs. In the latter case, they go at the speed of the slowest person.

Given that, with the torch, Dr. Jones can cross the bridge — in either direction — in one minute, the girl in two, the wounded guy in four, and the kid in eight, what are all the ways, if any, in which they can all survive?

You will solve this using both a direct approach and a synchronised product. You can and probably should use `wolf.py` as a template, since the problem is quite similar.

Note: there is a method `NFA.trim`, which removes all "useless" states (neither accessible nor co-accessible). It may be useful.

(11) Variant:

In another universe, Dr. Jones can cross the bridge in one minute, the girl in two, the wounded guy in *five*, and the kid in *ten*. Thus they need an extra three minutes of crossing time across all members. However, they only have two extra minutes to cross, for a total of *17 minutes*.

Can they still make it? How? Why?

(12) The Wolf, the Goat, the Cabbage, the Stick, the Fire, and Lulu:

I hope you liked *the Wolf, the Goat, and the Cabbage*, for they are now joined by an all-star cast of new and quirky characters in this high-octane sequel/reboot.

Lulu, the farmer, needs to get everyone on the other side of the river. Unfortunately, her boat only has room for three, and, if left unsupervised on either bank, the goat eats the cabbage, the wolf eats the goat, the wooden stick beats the wolf to death, and the fire burns the stick to ashes.

Solve that using a synchronised product.

How many solutions, if any, are there in total?

(13) The Thief, the Cop, Mom & Dad, two Boys, two Girls, and Why Don't You Just Throw In the Kitchen Sink, While You're At It?

You can safely skip this exercise. It's the same as before, with more everything.

I *really* hope you liked the sequel to the reboot of *the Wolf, the Goat, and the Cabbage*, because here comes the spin-off to the prequel to the reboot, and we are going to blow all our budget on special effects and Scarlett Johansson (as the Mom).

We now follow a cast of eight likable main characters with richly developed backstories: the thief, the cop, Mom & Dad, and their underage boys and girls, two of each. In a completely unforeseen plot twist, they find themselves in the Caribbeans and need to go from one small deserted island to the other to save America from... bad guys? Somehow.

In nostalgic homage to the premise of critically acclaimed previous installments of the series, they only have a small rowboat, seating two people at most.

Of course, only adults can row the boat at all, but the thief cannot be trusted in the least: he must not be alone in the boat, or he will escape and leave everyone stranded. He will also take every opportunity to stab somebody – *anybody* – in the back, whether on land or sea, unless the cop is here to prevent it. He *can* be left entirely alone on either island, though, as there is nowhere to run and nobody to stab.

While the cop supervises the thief, Mom and Dad must supervise the interactions of their spouse with their children of the opposite sex. The thief and the cop wisely abstain from interfering in family matters.

The boys tend to be too... energetic... for Mom to handle; leaving her with either or both boys without Dad's soothing presence means headaches — again, whether on land or sea.

Conversely, both girls have perfected the art of weaponizing puppy-dog-eyes and guilt trips to extort dance lessons, ponies, and girly sundries from Dad. Mom is of course wise to those tricks, and will not leave either girl any opportunity, either waiting on an island or alone on a boat, to sweet-talk her mushy-brained husband into frivolous spending sprees. That's *her* job.

Can they all get to the other island without anybody escaping, getting stabbed, suffering from a splitting headache, or being coaxed into buying the gold-plated, diamond-studded, Pink Collector's Edition of all Disney Princesses movies ?

How many solutions, if any, are there in total? Would you watch that movie? Have you had enough of that type of problem yet? I know I have. Only a few more of those to go...

(14) The Worm, the Centipede, and the Grasshopper:

The three Amigos want to cross the river (a decidedly common wish these days!). A fallen leaf will have to suffice as their "boat". It is large enough to accommodate all three, but only strong enough to support 60g, and no more.

Given that the worm weighs in at a hefty 50g, the centipede at 30g, and the grasshopper at 20g, and that any of them can manoeuvre the leaf, what are all the possible ways, if any, in which this can be achieved?

You will use a synchronised product.

(15) The two cans:

A wise hermit demands that you bring him exactly four litres of water. You are given two cans, devoid of any markings. All you know is that one contains 3 litres when full, and the other 5 litres. Their dented, irregular forms make it impossible to reliably tell how much they contain when they are partially filled, and you don't even have a ruler or anything to measure, say, four fifths of a can. Thus they are either empty, full, or filled by a previously known or inferred quantity of water (say, if you filled the 3 litres can and emptied it into the empty 5 litres can, then you know the latter contains 3 litres; if you filled the 5-litres can, then used it to fill the 3-litres can, then you know you have 2 litres left in the 5-litres can).

You have access to a water tap, which you can use to fill either can at will. You can also empty either can into the sewer. There is no limit to how much water you can draw or throw away — which is not very ecologically conscious on the hermit's part... is he even all that wise?

Use a synchronised product to solve the problem. Don't rush to start coding, but think carefully about the systems involved and their behaviours.

What is the shortest solution?

(16) Semaphors and naïve processes:

Note: we are going to see this problem and the synchronised product model for it during the lectures, Sec. 10.2.7[p69]: "Semaphores: first contact". Then all that will be left is the Python implementation. That said, it could be very interesting for you to try your hand at this exercise before *the corresponding lecture, if you already did all the previous ones.*

```
def semaphor sem:
  sem = 1   # initialise: one instance of resource
  def P(sem): wait until atomic{ if sem > 0:  sem--; break }
  def V(sem): atomic{ sem++ }

def process P0:
  while True:
    # noncritical section
    P(sem)
    # critical section
    V(sem)

def process P1:
  while True:
    # noncritical section
    P(sem)
    # critical section
    V(sem)

exec P0, P1
```

Write and implement a model in terms of a synchronised product of systems, and check that the race condition is avoided.

(17) Peterson's algorithm:

Note: we are going to see this algorithm and the synchronised product model for it during the lectures. Then all that will be left is the Python implementation. That said, it could be very interesting for you to try your hand at this exercise before *the corresponding lecture, if you already did all the previous ones.*

Recall Peterson's algorithm for mutual exclusion:

```
def binary_vars:
  W0    := 0 # process 0 wants critical access
  W1    := 0 # process 1 wants critical access
  Turn  := 0 # Whose turn is this ?

def process P0:
  while True:
    # noncritical section
    W0    := 1
    Turn  := 1
    wait until W1 = 0 or Turn = 0
    # critical section
    W0    := 0

def process P1:
  while True:
    # noncritical section
    W1    := 1
    Turn  := 0
    wait until W0 = 0 or Turn = 1
    # critical section
    W1    := 0

exec P0, P1
```

The aim is to verify that mutual exclusion and bounded waiting are achieved. To do so, write and implement a model in terms of a synchronised product of systems.

4 Extra exercises involving rivers (from JMC's collection)

Just in case you run out of exercises. . .

(18) Trois missionnaires: Trois missionnaires et trois cannibales doivent traverser une rivière. Les trois missionnaires et un cannibale savent ramer. Ils ont une barque de 2 personnes. S'il y a d'un coté ou d'un autre de la rivière un nombre supérieur de cannibales que de missionnaires, les missionnaires se font manger. N'oublier pas de ramener la barque, personne ne doit se faire manger...

(19) Les quatre couples: Quatre couples sont tout juste fiancés : Annie avec Armand, Béatrice avec Bernard, Caroline avec Charles, et Delphine avec Denis. Ils veulent pique-niquer de l'autre côté de la rivière. Ils peuvent louer une barque, mais qui ne peut pas prendre plus de 2 personnes à la fois. Les hommes sont d'une jalousie terrible, et aucun ne veut laisser sa fiancée en compagnie d'un autre homme même en public, à moins que lui-même 1 ne soit présent. Armand ne souffrira pas de voir Annie avec Bernard en son absence. Il y a au milieu de la rivière une île qui peut servir d'étape pendant la traversée. Le problème est de savoir comment traverser la rivière par le nombre minimum d'allées et venues. Aller de la rive à l'île ou de l'île à la rive compte pour un voyage, de même que d'aller d'une rive à l'autre. Tout le monde sait ramer. La seule contrainte provient de la jalousie des hommes : aucun d'eux ne peut prendre le bateau lorsqu'une femme autre que sa fiancée est seule soit sur l'île, soit sur l'autre rive, même s'il a une autre destination. 17 voyages suffisent.

(20) Le missionnaire et les Indiens: Il y a 100 ans, un groupe de 3 missionnaires se frayait un chemin dans la forêt amazonienne en compagnie de 3 guides indiens. Arrivés devant une rivière, ils trouvèrent une pirogue qui ne pouvait transporter que 2 personnes à la fois. Elle était difficile à man ?uvrer, et ne pouvait l'être que par un seul des 3 Indiens, et par un seul des trois missionnaires. Les missionnaires ne se fiaient guère aux Indiens, et réciproquement les Indiens se méfiaient de la civilisation moderne. Les missionnaires firent donc tout ce qu'il faut pour n'être jamais moins nombreux que les Indiens sur l'une et l'autre des rives. Comment y parvinrent-ils pour un nombre minimal de traversées ?

5 CTL Verification

(21) Semaphors: CTL verification

Using the `ctl` Python module, check whether the following properties are satisfied by our earlier semaphor program — question 16[p25]:

- The processes shall never both be in their critical sections at the same time.
- No process shall starve. That is to say, both processes shall enter their critical section infinitely often.

(22) Peterson's algorithm: CTL verification

Using the `ctl` Python module, check whether the following properties are satisfied by Peterson's solution — question 17[p26]:

- The processes shall never both be in their critical sections at the same time.
- Any process that wants to enter its critical section shall eventually do so.
- Processes alternate their accesses to their critical section. That is to say, if $\mathtt{P}_0$ has just accessed its critical section, then the next one to access its critical section must be $\mathtt{P}_1$, and vice-versa.

Part II

Lecture Notes: Formal Verification

6 Meta-information about the course

6.1 Note on the notes:

These lecture notes are a **work in progress**, and have no ambition, even in the fullness of time, to be as complete and self-contained as my Python lecture notes. At this point, those are more of a Python *book* that entirely replaces the lectures, at the benefit of more lab classes.

In this Verification class, however, lectures are and will remain necessary due to the more abstract nature of the content. Those burgeoning notes shall serve mostly as slides during the lectures, and as memento afterwards. You are encouraged to take your own notes in addition.

Like my Python lecture notes, and for the same reason, the document may alternate between French and English, with a will to converge towards the latter.

Acknowledgements: This course is partly based upon the lectures given at INSA CVL for many years by Jean-Michel Couvreur, until I took it over in 2019–2020. Books, lecture notes, and slides by J.-P. Jouannaud, Joost-Pieter Katoen, Yohan Boichut, Patricia Bouyer, and many others, provided sundry examples and elements of inspiration. Any mistakes in this document are, likely, mine.

6.2 Course prerequisites and student assessment

Prerequisites:

- Basic set theory: set comprehension notation, powersets, functions and relations as sets, etc.
- Formal Language Theory: basic notions of NFA, DFA, their languages, determinisation, how to compute $A \cap B$ (synchronised product), etc.
- You will have to get familiar with my NFA framework (Python). See the section on lab classes. Therefore a decent practice of Python is a prerequisite, not at all for the theoretical aspects of the course, but by dint of being necessary to solve the practical problems. Students who did *not* follow the Python course with me last year would be well advised to check my Python lecture notes[c] and get up to speed if necessary.
- Basic mastery of Linux-based OS. My framework also works under Windows, but I develop and test under Linux only. Use Windows at your own risk.

Assessment: final examination, on paper.

[c] Available on `https://celene.insa-cvl.fr/course/view.php?id=208`.

7 Introduction: What is Formal Verification?

7.1 Problem: Disaster stories

(1) 1985-86: Therac 25: « x Up Edit e Enter » in less than 8s -> race condition (compétition / concurrence critique) 125x the radiation. Fatal overdoses 86

(2) 1990: AT&T: bug in switch / break (C), race condition : no phone for 9h on whole USA east coast; N*100 M

(3) 1994: Pentium Floating Point Division bug: 470 M ; 1 in 9 billion results were flawed; all procs replaced

(4) 1996: Ariane 5 flight 501, 500M, explodes after 37s. A data conversion from a 64-bit floating point to 16-bit signed integer

(5) 1999: Mars climate orbiter, SI units N.s vs non SI pound.s 328 M

(6) Year 2000: $457 billion

(7) 2008: Heathrow Terminal 5 Opening new baggage handling system: tested with 12,000 test pieces of luggage before opening. Turned out that it couldn't handle a passenger manually removing a bag. 42 000 bags misplaced, 500 flights cancelled.

(8) 2012: Knight Capital Group: $440 M lost in 30 mins due to buggy trading software. There were all sorts of bad coding practices at play here. Interesting link about this.

(9) . . .

7.2 Solution: Verification

(1) What is verif? Duality program / specification: do they match? If not prog *or spec* might be erroneous: verif adequation, correct, try again. Iterative.

(2) Spec is what we use in our head all the time; can be more or less formal. We are interested in mathematical proofs, thus the spec needs to be formal.

(3) Do we verify *the program itself*, or a model or abstraction thereof? Can't check the compiler, the OS, the ambient temperature,. . . always abstract that which seems irrelevant, and hope it *is* irrelevant in practice.

See Ken Thompson's Turing award lecture: *Reflections on trusting trust* [Thompson, 1984].

(4) A vast domain, with many techniques:

a. Testing (how are test cases generated? Can be from spec? coverage? Does 100% coverage mean 100% correctness?), Test Driven Development (TDD)

b. Proof (Hoare logic,...), not adapted to reactive systems or concurrency.

$$\{P\}C\{Q\}, \qquad \frac{\{B \wedge P\}S\{Q\} \quad , \quad \{\neg B \wedge P\}T\{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}}$$

automated to a large degree.

c. Curry-Howard certified prog (B, Coq,...),

d. Abstract interpretation (simulate exec with approximation / bounds),

e. **Model-Checking**, good for concurrency, can be

f. . . .

7.3 A Brief History of Program Proof

(1) 1949 Turing proposes mathematical proof of programs

(2) 1969 Hoare logic for sequential programs; first order

(3) 1975 Constat : vérif. inadaptée à systèmes réactifs

(4) 1977 Pnueli propose d'utiliser les logiques temporelles for concurrent programs

(5) 1981 Model checking de CTL par Clarke Emerson, Sifakis et al.

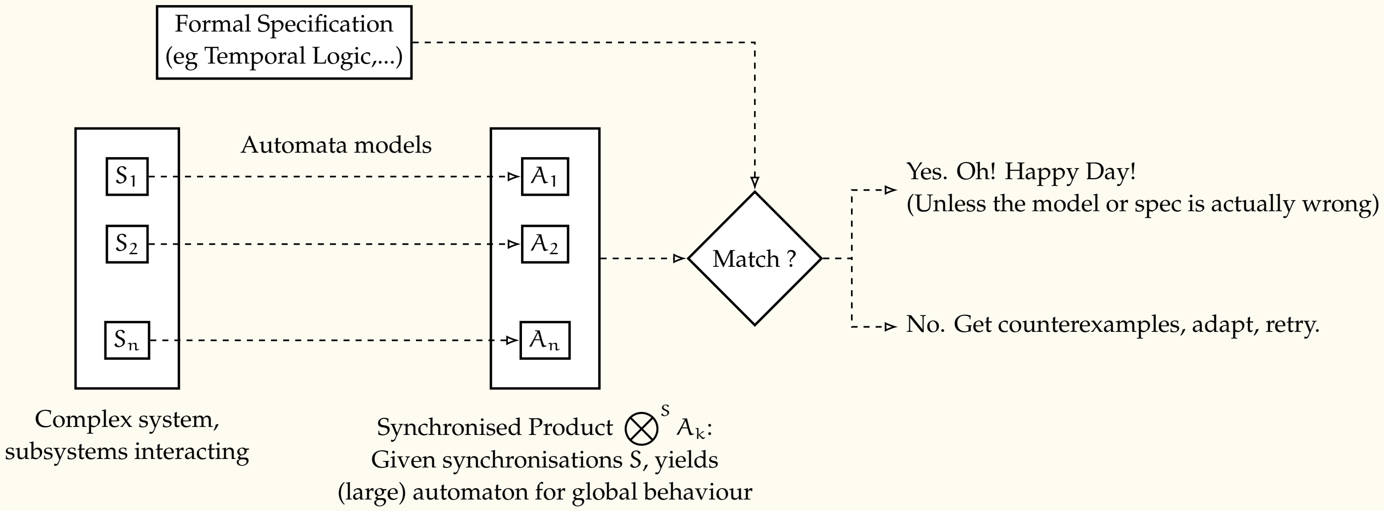
(6) 1980-1990 Nombreux résultats théoriques

(7) 1990-2000 Énorme amélioration des performances, Extensions : proba, temps,...

(8) 2000-... MC adopté par les principaux fondeurs (Intel, etc.)

(9) 2008 Prix Turing décerné à Clarke, Sifakis et Emerson

7.4 Our focus in this course: Model Checking


Formal Specification
(eg Temporal Logic,...)
Automata models
S_1
S_2
S_n
A_1
A_2
A_n
Match ?
Yes. Oh! Happy Day!
(Unless the model or spec is actually wrong)
No. Get counterexamples, adapt, retry.
Complex system,
subsystems interacting
Synchronised Product $\bigotimes^S A_k$:
Given synchronisations S, yields
(large) automaton for global behaviour

Examples of applications of model-checking in "the real world":

- Famously, [Lowe, 1996] broke and fixed the Needham-Schroeder public-key protocol, revealing mistakes that had remained undiscovered for over 17 years.
- [Clarke et al., 1993], analysing a model of over 10^{30} states, found mistakes in the IEEE Futurebus+ industry standard, leading to a substantial revision of the protocol.
- [Staunstrup et al., 2000] successfully verified a train model of over 1421 components, for a total state space of 10^{476}.
- Widely used for hardware and software verification at IBM, Intel, Microsoft, NASA (Mars Pathfinder, Deep Space-1),... wherever there are critical systems and lots of money on the line.

7.5 Provisional course plan

(1) modélisation par systèmes états transitions / structures de Kripke / automata

(2) produits synchronisés de systèmes

(3) problèmes d'accessibilité; algorithme de Peterson

(4) logiques de mots finis et infinis: logiques monadiques faible et forte du second ordre d'un successeur (w)S1S, logiques de temps arborescent et linéaire: CTL*, CTL, LTL.

Last year we stopped there.

(5) algorithme de vérification CTL

(6) automates de Büchi et algorithme de vérification LTL (en fonction du temps)

(7) Logique de Hoare (dernier cours)

8 State Systems and Modelisation

8.1 Brief Reminders About Nondeterministic Finite State Automata

Un **automate fini non déterministe (NFA)** est un 5-uplet $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$ où:

- Q: ensemble fini d'états
- Σ: alphabet fini
- $I \subseteq Q$: états initiaux
- $F \subseteq Q$: états terminaux
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$: relation de transition

$$(p, a, q) \in \Delta \overset{\text{notation}}{\equiv} p \xrightarrow{a} q \qquad \Delta(p, a) = \{q \mid p \xrightarrow{a} q\}$$

We shall not hesitate to use object-like notation; for instance, if $A \in \mathrm{NFA}$, we can write $A.Q$ for its set of states. This is not a classical notation, but it avoids having to define unique names for each component when we have several automata to deal with, and matches the notations in my Python NFA framework.

Clôture transitive des transitions & sémantique

Soient $u, v \in \Sigma^*$. Si $p \xrightarrow{u} q$ et $q \xrightarrow{v} r$ alors $p \xrightarrow{uv} r$.

Sémantique: langage reconnu par un état:

$$[\![q]\!] = \{u \in \Sigma^* \mid \exists q_{\mathrm{ini}} \in I : q_{\mathrm{ini}} \xrightarrow{u} q\}$$

Sémantique: langage reconnu par un automate:

$$[\![A]\!] = \bigcup_{q_{\mathrm{fin}} \in F} [\![q_{\mathrm{fin}}]\!] = \left\{ u \in \Sigma^* \;\middle|\; \exists q_{\mathrm{ini}} \in I, q_{\mathrm{fin}} \in F : q_{\mathrm{ini}} \xrightarrow{u} q_{\mathrm{fin}} \right\}$$

Déterminisme, non-déterminisme et ε

Exercice: $abc \in [\![A]\!] \overset{?}{\iff} \exists q_{\mathrm{ini}} \in I; p, q, r \in A.F : q_{\mathrm{ini}} \xrightarrow{a} p \xrightarrow{b} q \xrightarrow{c} r$

Non, car $abc = a\varepsilon bc = \varepsilon ab\varepsilon c = \cdots$

Non-déterminisme: ε-transitions, multiples états initiaux, et "choix" $p \xrightarrow{a} q$ et $p \xrightarrow{a} q'$.

Un automate est **déterministe** si:

- $|I| \leqslant 1$
- La relation de transition δ est une fonction partielle $Q \times \Sigma \longrightarrow Q$

Etats accessibles, coaccessibles, trim (émondage)

Complete automata (not to be confused with complementation)

8.2 On machine: `lecture_automata_products.py`

Practice up to (and including) the section on determinisation.

More on that later (products)

Note: You will use this framework in lab classes. Note that there is some documentation for it in Sec. 2[p9]: "Basic finite state systems".

8.3 Modelling through automata: generalities

A shift on philosophy regarding automata:

Last year: descriptors for languages; internal details such as number of state ultimately unimportant

This year: outil de modélisation des états du système.

Langage généralement moins important, états et transitions modélisent des aspects réels.

Exemples types de problèmes:

(1) Systèmes à états finis: montre hh:mm, digicode...

(2) Loup chèvre et chou | Wolf, goat and cabbage

(3) Acteurs en concurrence, qui coopèrent / se synchronisent sur certaines choses, devant respecter certaines contraintes

(4) Exclusion mutuelle, sémaphores, algo Peterson,...

(5) Sureté (toujours / jamais)

(6) Vivacité (un jour, fatalement)

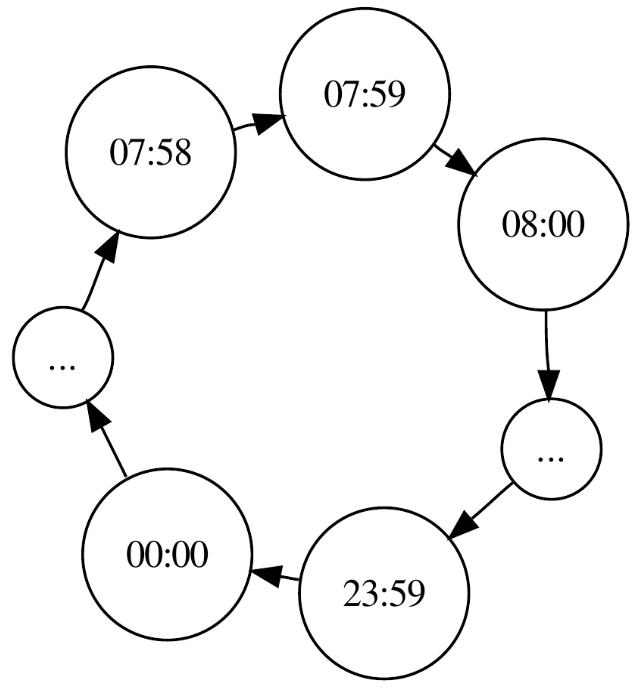
8.4 Examples of Isolated Systems

Let us see a few examples of simple systems, and not-so-simple systems that we still tackle by viewing them in their globality, at least for now — we will move towards a "product of sub-systems" soon enough.

8.4.1 Digital Clock

Consider a digital clock $\mathrm{hh} : \mathrm{mm}$ (24h). How many states?

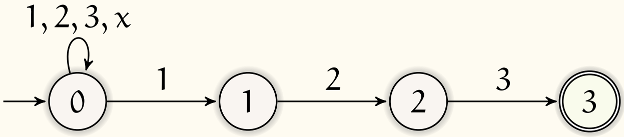
$$24 \times 60 \quad = \quad 1440$$



8.4.2 Digicode, pass 123

You first lab class exercise will be to implement this.

x stands for any digit other than 1, 2, 3.



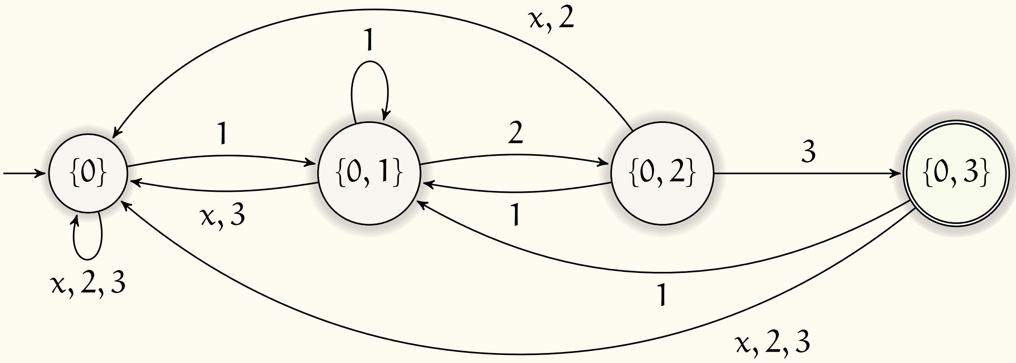
Déterminiser, et écrire une version déterministe directement. Rappels de l'année dernière !

END FIRST LECTURE (2020–2021)

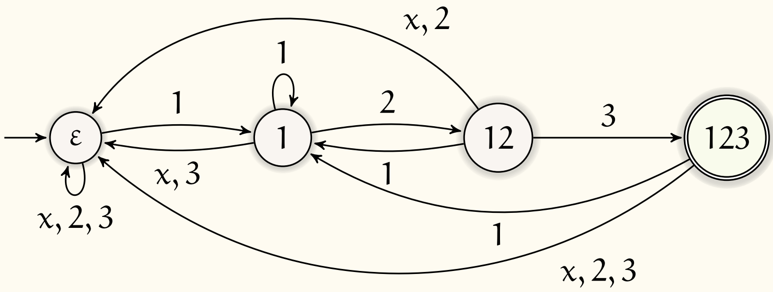
		1	2	3	x
Initial	**0**	0, 1	0	0	0
	1		2		
	2			3	
Final	**3**				

Determinisation algorithm: preferably use table:

		1	2	3	x
Initial	$\{0\}$	$\{0,1\}$	$\{0\}$	$\{0\}$	$\{0\}$
	$\{0,1\}$	$\{0,1\}$	$\{0,2\}$	$\{0\}$	$\{0\}$
	$\{0,2\}$	$\{0,1\}$	$\{0\}$	$\{0,3\}$	$\{0\}$
Final	$\{0,3\}$	$\{0,1\}$	$\{0\}$	$\{0\}$	$\{0\}$



To write a direct DFA, reason in terms of "how much of the correct passcode have I seen yet?".

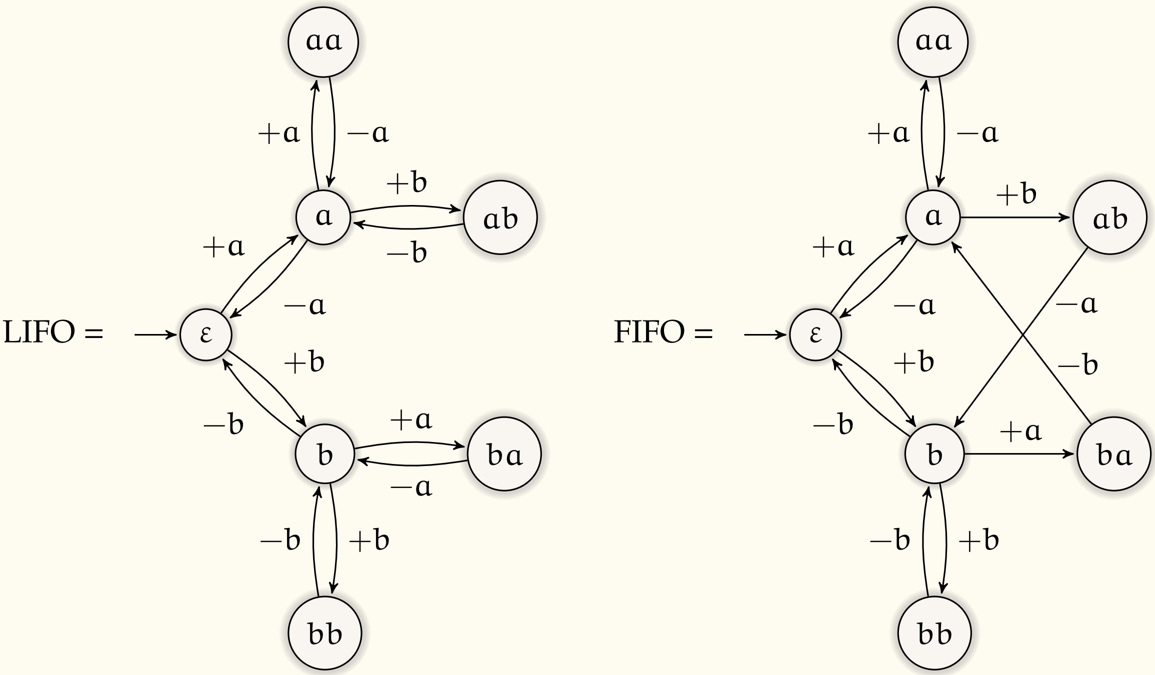


END FIRST LECTURE (2019–2020)

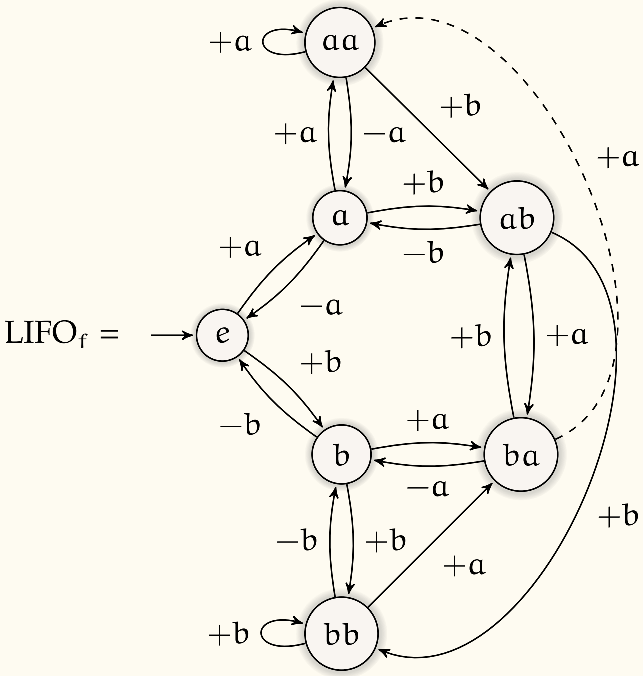
8.4.3 LIFO (Stack) and FIFO (Queue) of size 2

$\Sigma = \{ a, b \}$, input denoted by $+$, output by $-$.

Maximum storage capacity of 2 symbols.



Suppose now that, instead of being unable to add new symbols past capacity, we simply forget the oldest stored symbol. We get the following behaviour:



8.4.4 Incrementable Integer Variable

So far we have only seen automata with a given, constant, number of states and transitions. Let us get used to defining potentially infinite collections of automata as a function of some parameters. The following was an exercise given in the 2019–2020 final exam, and will be very useful for several problems that involve counting.

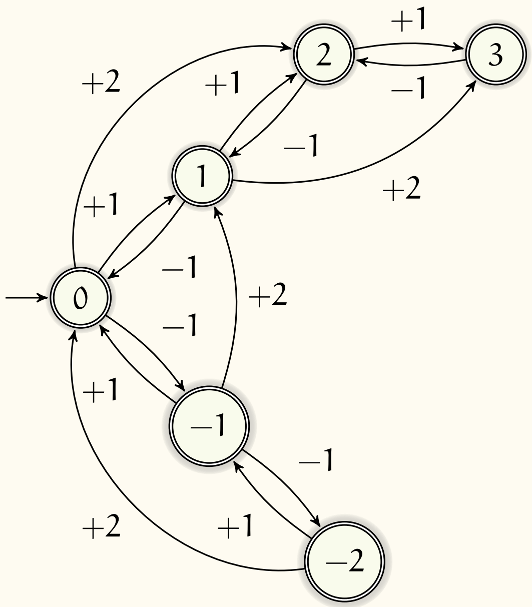
For all $n, m, i \in \mathbb{Z}$ and $X \subseteq \mathbb{Z}$, formally define a NFA $V(n, m, i, X)$ representing an integer variable on the interval $[\![n, m]\!]$, initialised to i, that can be incremented by the quantities in X, and only by those.

(We speak of *decrementation* when the quantity by which we increment happens to be negative.)

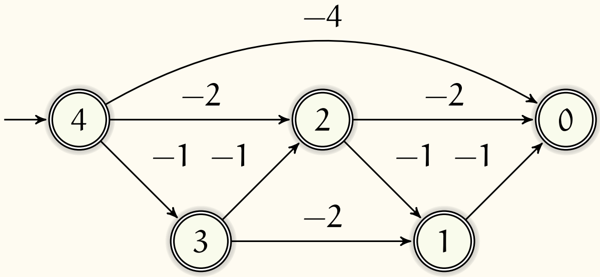
We permit neither overflow nor underflow. We consider all states final.

For instance, we have:

$V(-2,3,0,\{-1,1,2\}) \;=$



$V(0,4,4,\{-1,-2,-4\}) \;=$



The formal definition of V is:

$$V(n,m,i,X) \;=\; \left[\begin{array}{lcl} \Sigma & = & X \\ Q & = & [\![n,m]\!] \\ I & = & \{i\} \\ F & = & Q \\ \Delta & = & \left\{ p \xrightarrow{x} q \;\middle|\; p,q \in Q,\ x \in X,\ q = p + x \right\} \end{array}\right]$$

It can easily be implemented in Python; for instance:

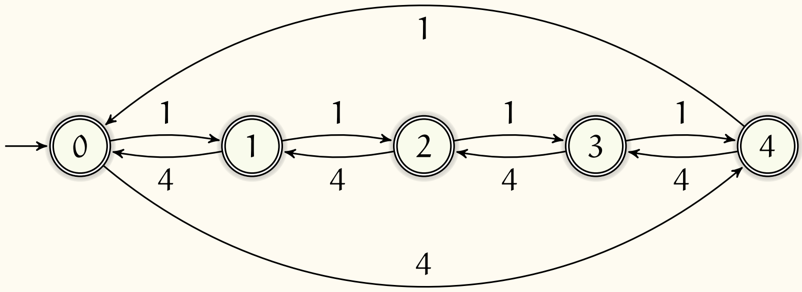
```
def increment(n,m,i,X):
    return NFA({i}, Q := range(n, m + 1), {
        (p, x, p + x )
        for p in Q for x in X
        if n <= p + x  <= m # p+x in Q would be clearer but less efficient
    }).named(f"Increment({n}, {m}, {i}, {X})")
```

9 Incrementable Unsigned Integer Variable with Overflow

For all $n, i \in \mathbb{N}$ and $X \subseteq \mathbb{Z}$, formally define a NFA $V_o(n, i, X)$ representing an integer variable on the interval $[\![0, n]\!]$, initialised to i, that can be incremented by the quantities in X, and only by those. That variable is subject to integer overflow with *wrap around* semantic, like `unsigned char/int` in C, meaning that whenever the variable exceeds its maximum, it wraps back around to 0.

We consider all states final. For instance, we have

$V_o(4, 0, \{1, 4\}) \quad =$



The formal definition of V_o is:

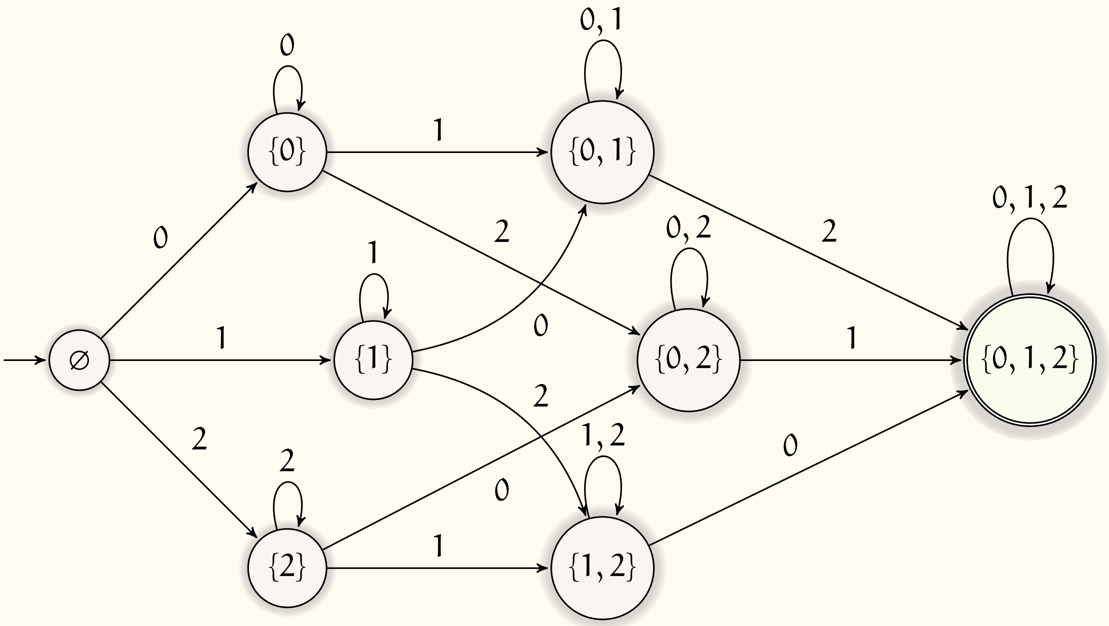
$$V_o(n, i, X) \quad = \quad \left[\begin{array}{lcl} \Sigma & = & X \\ Q & = & [\![0, n]\!] \\ I & = & \{i\} \\ F & = & Q \\ \Delta & = & \left\{ p \xrightarrow{x} q \;\middle|\; p, q \in Q,\ x \in X,\ q = p + x \mod n + 1 \right\} \end{array}\right]$$

10 Set Variable

For any set X, formally define an NFA $S(X)$ representing a set variable, initially empty, to which elements of X can be added. The automaton shall be in an accepting state when the variable is "full", that is to say, when all elements of X are already present. Following set semantics, elements that are already present in the set variable can still be added to it.

For instance, we must have:

$S(\{0,1,2\}) =$



The formal definition is:

$$S(X) = \left[\begin{array}{lcl} \Sigma & = & X \\ Q & = & \wp(X) \\ I & = & \{\varnothing\} \\ F & = & \{X\} \\ \Delta & = & \left\{ p \xrightarrow{x} p \cup \{x\} \;\middle|\; p \in Q,\ x \in X \right\} \end{array}\right]$$

10.0.1 FIFO / LIFO(n, m)

Following the same principle as above, we can generalise FIFO and LIFO to arbitrary numbers of symbols and capacities. For all $n, m \in \mathbb{N}^*$, let us model the finite-state behaviour of a LIFO / FIFO on symbols in $\mathbb{A}$, with $|\mathbb{A}| = n$, of capacity m — that is to say, capable of storing m symbols.

$$\mathrm{LIFO}(n, m) = \left[\begin{array}{lcl} \Sigma & = & \left\{ \pm a \;\middle|\; \pm \in \{+, -\},\ a \in \mathbb{A} \right\} \\ Q & = & \mathbb{A}^{\leqslant m} \quad = \quad \bigcup_{k=0}^{m} \mathbb{A}^k \\ I & = & \{\varepsilon\} \\ F & = & \varnothing \\ \Delta & = & \left\{ p \xrightarrow{+a} pa,\ pa \xrightarrow{-a} p \;\middle|\; a \in \mathbb{A},\ p, pa \in Q \right\} \end{array}\right]$$

Similarly,

$$\mathrm{FIFO}(n,m) = \left[\begin{array}{lcl} \Sigma & = & \left\{ \pm a \;\middle|\; \pm \in \{+,-\},\ a \in \mathbb{A} \right\} \\ Q & = & \mathbb{A}^{\leqslant m} = \displaystyle\bigcup_{k=0}^{m} \mathbb{A}^k \\ I & = & \{\varepsilon\} \\ F & = & \varnothing \\ \Delta & = & \left\{ p \xrightarrow{+a} pa,\ ap \xrightarrow{-a} p \;\middle|\; a \in \mathbb{A},\ p, pa, ap \in Q \right\} \end{array}\right]$$

Implementing this is a lab class exercise.

To obtain a forgetful behaviour in either FIFO or LIFO, one need only add the following transitions to Δ:

$$\left\{ ap \xrightarrow{+b} pb \;\middle|\; a, b \in \mathbb{A},\ p \in \mathbb{A}^*,\ ap \in Q,\ |ap| = m \right\}.$$

10.0.2 The Wolf, the Goat, and the Cabbage (WGC)

Alert: *a critical and, judging by last year's final exam performance,* difficult *section of the class begins here. (For Dark Souls players among you, imagine this point as the fog wall leading to a boss room, with countless bloodstains littered on the floor. Pay attention.)*

This is a famous folklore problem, dating back to the 9th century, at least. The following description is shamelessly copied from Wikipedia:

> *Once upon a time a farmer went to a market and purchased a wolf, a goat, and a cabbage. On his way home, the farmer came to the bank of a river and rented a boat. But crossing the river by boat, the farmer could carry only himself and a single one of his purchases: the wolf, the goat, or the cabbage.*
>
> *If left unattended together, the wolf would eat the goat, or the goat would eat the cabbage.*
>
> *The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact. How did he do it?*

Though it does not look like it, this type of problem — there are infinite variations on that theme — is "just" a concurrent systems problem in funny clothes, which we shall use to refine our intuitions on how to model that kind of things with automata.

Let us first solve that by building an automaton representing the relevant aspects of all the possible reachable configurations of these entities. We shall see the whole scene as a single, large system, using our state space to store the relevant information.

We shall refer to this solution as the *naïve* approach. Of course a 9th century scholar might not have thought of *any* solution involving automata theory as “naïve”; this is not an absolute judgement value. . . By this I just mean that we shall come back to this problem later — once we have introduced the theoretical background of synchronised products — and do it again in a *much* cleaner way.

Let us move on to the naïve model.

Let A be the set of actors / entities whose status we need to track: The Wolf, the Goat, and the Cabbage, as well as the Farmer. No need for the Boat, because it is wherever the Farmer is at any time, and thus would be redundant.

$$A = \{ W, G, C, F \}$$

What do we need to keep track of? The relevant info is on what bank of the river each actor is. Let us call the banks left and right, or 0 and 1. That information is a function of type

$$A \to \{ 0, 1 \}$$

which can more compactly be represented by a set, e.g. the set of all actors on the left bank. This is the view we take. Thus we let

$$Q \subseteq \wp(A)$$

We use $\subseteq$ here rather than $=$ because presumably not all 2^4 configurations / states may be reachable. Indeed, we specifically must not allow certain actors to remain unattended — by the farmer — on the same bank. We’ll come back to that shortly.

Our initial states are, assuming everyone starts on the left bank,

$$I \quad = \quad \{ A \} \, ,$$

and our final states are

$$F \quad = \quad \{ \varnothing \} \, .$$

Let us write a predicate to define which collections of actors on the same bank are licit with respect to the constraints of the problem: in English, if Wolf and Goat, or Goat and Cabbage, are together, then the Farmer must be with them. Let $s \subseteq A$ be a collection of actors; they are licit on the same bank if

$$\ell_0(s) \quad = \quad \big[\{ W, G \} \subseteq s \ \vee \ \{ G, C \} \subseteq s \big] \implies F \in s \, .$$

Recalling that a state q is the collection of actors on the left bank, and that $\overline{q} = A \setminus q$ is the corresponding collection of actors on the right bank, a state is licit if both left and right bank are:

$$\ell(q) \quad = \quad \ell_0(q) \ \wedge \ \ell_0(\overline{q}) \, .$$

Let us note that our initial state is licit. If that weren't the case, the problem would have no solution.

There remains to build the transitions and reachable licit states. First, left-to-right movements:

$$\Delta \ni p \xrightarrow{\lambda} q \quad \text{if} \quad \begin{cases} p \in Q,\ \ell(p),\ \ell(q) & \\ p \ni F, a & \textit{not necessarily distinct, Farmer may be alone} \\ \lambda = \{F, a\} & \textit{arbitrary label; it makes sense to track who moves} \\ q = p \setminus \{F, a\} & \textit{they went to the right, so they are not on the left} \end{cases}$$

Then, right-to-left movements:

$$\Delta \ni p \xrightarrow{\lambda} q \quad \text{if} \quad \begin{cases} p \in Q,\ \ell(p),\ \ell(q) & \\ \overline{p} \ni F, a & \textit{they are on the right} \\ \lambda = \{F, a\} & \\ q = p \cup \{F, a\} & \textit{they return to the left} \end{cases}$$

There are no other transitions. That is to say, Δ is the smallest set, with respect to inclusion, that satisfies those rules.

This gives us a recipe for gradually building new transitions and new states, starting from our initial state.

$\ell(p)$ could be removed as a condition, so long as we explicitly test that initial states are licit; then the $\ell(q)$ condition in the rules will ensure, inductively, that all generated states are licit.

This is implemented in `wolf.py`. Take a look.

10.0.3 WGC, states as functions rather than "left-bank" sets

We *could* have kept functions $A \to \{0, 1\}$ for states, and obtained a slightly different, equivalent model. Perhaps a little bit more difficult to follow, to use, and to implement because it requires heavy use of inverse functions, and operators to add, overwrite, or remove pairs $x \mapsto f(x)$ from functions.

It *may* be interesting to write it out, but I haven't done so for now. It runs the risk of confusing students further, with *three* different versions of this solution in total: naïve, naïve with functions, and product-based.

END FIRST LECTURE

10.0.4 Indiana Jones and the Temple of Verification

Here is another funny problem; we shall do the math together, but this time you will have to implement it yourself in lab class. Of course we use the naïve approach for now, as it is the only one we know.

> *Indiana Jones, his annoying girlfriend, a wounded guy, and a whiny kid find themselves in a dire predicament: savage cannibalistic cultists are on their heels; in 15 minutes, they will be toast. . .* or on *toast.*
>
> *Their only hope? swiftly crossing the crocodile-filled ravine, using the threadbare, rickety bridge. It is quite clear that the bridge can only support the weight of two persons at most — even if one of them is a kid.*
>
> *To make things worse, night has fallen, and the bridge is far too treacherous to walk blind; a torch is necessary to examine the worm-eaten planks* before *setting foot on them.*
>
> *Dr. Jones, being a seasoned adventurer, does have a torch in his inventory; the group will have to find a way to share. Though nobody else has a torch of their own, all can use Dr. Jones' torch to cross the bridge on their own or in pairs. In the latter case, they go at the speed of the slowest person.*
>
> *Given that, with the torch, Dr. Jones can cross the bridge — in either direction — in one minute, the girl in two, the wounded guy in four, and the kid in eight, what are all the ways, if any, in which they can all survive?*

The Lamp is very much a relevant actor, as you cannot cross the bridge without it. It plays the same role as the Farmer / the Boat in WGC, except that there are no constraints on the "banks" in this problem. Moreover, unlike the Farmer, the Lamp cannot cross under its own power. This is a consideration for the Δ, however. We have the actors:

$$A = \{ I, G, W, C, L \}$$

For our states, again we need to track left/right positions for each actor (coded as set of actors on left side), but this time we have a time component, a decreasing timer, starting at 15 minutes.

$$Q \subseteq \wp(A) \times [\![0, 15]\!] \tag{10.1}$$

We begin with everybody on the left side, and 15 minutes before the cannibals arrive.

$$I = \{ \langle A, 15 \rangle \}$$

We succeed if at any point everybody has crossed safely, regardless of how many minutes we have left to kill.

$$F = \{\emptyset\} \times [\![0, 15]\!]$$

Of course we would be surprised if most of those states were actually reachable (everybody crosses in 0 minute? Optimistic!) but who is to say you can't cross with, say, 5 minutes left? Let's simply not get into that at that point, and accept everything.

Now, on to the transition rules. We need a way to keep track of the time cost associated to certain groups of people crossing the bridge. Let's dedicate a function to that task:

$$\tau \;=\; \left|\begin{array}{r} I \mapsto 1 \\ G \mapsto 2 \\ W \mapsto 4 \\ C \mapsto 8 \\ L \mapsto 0 \end{array}\right. \quad \text{and} \quad \forall s \subseteq A,\ \tau(s) = \max_{a \in s} \tau(a)$$

Now we have everything to translate the rules of the problem into automata transitions:

We have

$$\Delta \ni \langle p, t\rangle \xrightarrow{\lambda} \langle q, t'\rangle \quad \text{if} \quad \begin{cases} p \supseteq \underbrace{\{L, a, b\}}_{\lambda},\ a \neq L \\ q = p \setminus \lambda \\ t' = t - \tau(\lambda) \geqslant 0 \end{cases} \quad \text{or} \quad \begin{cases} \overline{p} \supseteq \underbrace{\{L, a, b\}}_{\lambda},\ a \neq L \\ q = p \cup \lambda \\ t' = t - \tau(\lambda) \geqslant 0 \end{cases}$$

We could attempt to optimise in various ways, for instance reasoning that we want to accumulate people on the right bank, requiring $|\lambda| = 3$ going right and $|\lambda| = 2$ going left, but that's anticipating the solution through intuition. That is generally a bad idea, because it is the system's job to find a solution, *all* solutions, and it is faster and more reliable at it than you are.

Each time you deviate from just modelling the problem faithfully, you risk introducing mistakes or cutting off paths that may lead to solutions or to interesting insights about the system. Only do so to deliberately cut off branches that do not interest you or help the system / make its output more readable if the problem has too many states.

END SECOND LECTURE

10.1 A Taxonomy of Automata Products

Let us go back to the theory, and define the notions of products that we need to model the interaction of concurrent sub-systems / actors / etc in a systematic way. We shall then immediately come back to the WGC problem and compare the naïve and product solutions.

A **automaton product** is any operator

$$\odot : \mathrm{NFA}^* \to \mathrm{NFA} \qquad \text{such that} \qquad \Big(\bigodot_k A_k\Big).Q \;\subseteq\; \prod_k A_k.Q\,,$$

that is to say, the set of states of an automaton product is (a subset of) the **cartesian product** of the set of states of the input automata. In practice, we also tend to have

$$\Big(\bigodot_k A_k\Big).I \;=\; \prod_k A_k.I\,. \tag{10.2}$$

Let us think about what this means. If we have n automata $A_1, \dots, A_n$, then a state of $B = \bigodot_{k=1}^{n} A_k$ is a tuple $\langle q_1, \dots, q_n \rangle$ where $q_k \in A_k.Q,\ \forall k$. That is to say, B is the *global system*, where we keep track of all sub-systems. If (10.2) holds — it generally does — then the global system starts with each sub-system being in its own initial state, which makes sense.

Therefore **an automaton product is any operation through which we unite sub-systems into a larger system**.

You may have noticed that, so far, we have only touched on the state spaces, *maybe* on initial states, and not at all on the *behaviour* of the product: we have no transitions, no final states, etc. This is intended; products are a *class* of operations, not a specific operation, because there are infinitely many legitimate ways to "glue" sub-systems together. Maybe we want them to work in complete lockstep, with each action being a synchronised effort of all members; maybe we need them to be completely independent and ignore each other; maybe some of them should synchronise on some symbols and not on others, in which case, on *which* symbols do *which* subsystems synchronise?

All of this depends entirely on what kind of problem we need to solve, and what phenomenon we are modelling.

We shall now review different kinds of common products, some of which you will have seen last year in automata theory, and finally generalise the concept into the **parametric synchronised product**, which shall become our keystone for the rest of the course.

For the sake of clarity and notational convenience most products will be defined as binary operators. Generalising the notions to n-ary operators is left as a — mostly tedious and pointless — exercise to the reader. It *can* nevertheless be a good idea to do so if you are not at ease with set theory and are struggling to understand the n-ary products defined later in this course.

10.1.1 Fully Synchronised Product $\otimes$

This is by far the most commonly encountered; it is used to prove that regular languages are closed under $\cap$ — although there is another proof based on the equality $L \cap M = \overline{\overline{L} \cup \overline{M}}$.

10.1.1.1 Fully Synchronised Product $\otimes$ for $\cap$

It has the property $[\![A \otimes B]\!] = [\![A]\!] \cap [\![B]\!]$. The idea is that both automata read the same word together; that is to say, for each letter read, both automata simultaneously take their transitions, reading the same letter, and both must accept the word in order for the product to accept it. Then the word is accepted by $A \otimes B$ if and only if both A and B accept it. We remove any ε-transitions if necessary, and take:

$$A \otimes B \;=\; \left[\begin{array}{lcl} \Sigma & = & A.\Sigma \cap B.\Sigma \\ Q & \subseteq & A.Q \times B.Q \\ I & = & A.I \times B.I \\ F & = & A.F \times B.F \\ \Delta & = & \left\{ \langle p, p' \rangle \xrightarrow{a} \langle q, q' \rangle \;\middle|\; \begin{array}{l} p \xrightarrow{a} q \in A.\Delta \\ p' \xrightarrow{a} q' \in B.\Delta \end{array} \right\} \end{array}\right]$$

We have indeed $[\![A \otimes B]\!] = [\![A]\!] \cap [\![B]\!]$. Implementing this in Python is very straightforward; here is the code in my NFA framework:

```
def __and__(s,o):
    """fully synchronised product: language intersection"""
    if not all( s.Σ ):s = s.rm_eps()
    if not all( o.Σ ):o = o.rm_eps()
    return NFA(
        { (i,j) for i in s.I for j in o.I },
        { (i,j) for i in s.F for j in o.F },
        { ((p,P), a, (q,Q))
          for (p,a,q) in s.Δ for (P,b,Q) in o.Δ if a == b },
        name=f"({s.name} ∩ {o.name})")
```

Demo in `lecture_automata_products.py`.

10.1.1.2 *Fully Synchronised Product $\oplus$ for $\cup$*

Although it is rarely used, the same construction, with very minor changes, can be repurposed for $\cup$. Of course, simply taking the disjoint union of every component of the input NFA is far simpler and cleaner, but yields NFA in all cases, even if the inputs were DFAs. The product construction, on the other hand, does preserve determinism.

The idea here is almost the same as before, both automata A and B read the same word in lockstep together; the $A \oplus B$ accepts the word if *either* of them accepts it. It looks as though the only difference compared to $\otimes$ is *either* instead of *both*. This would translate simply to a change in the final states of the product.

But there is a trap, here. Suppose $[\![B]\!] = \emptyset$. Then you want $[\![A \oplus B]\!] = [\![A]\!] \cup \emptyset = [\![A]\!]$, right? But suppose B is written simply as having no transitions; this is a valid way to write an empty automaton. Then how can A and B *both* read a word in $[\![A]\!]$, when B has no rule to do so? If we applied the product of rules as above, $A \oplus B$ would itself have no rules at all, and accept the empty language.

The solution here is to **complete** the automata on $A.\Sigma \cup B.\Sigma$, to make sure they both can actually read the words. Of course, ε-transitions are still unwelcome. Let us assume now that A and B are complete. We have:

$$A \oplus B = \left[\begin{array}{lcl} \Sigma & = & A.\Sigma \cup B.\Sigma \\ Q & \subseteq & A.Q \times B.Q \\ I & = & A.I \times B.I \\ F & = & A.F \times B.Q \ \cup \ A.Q \times B.F \\ \Delta & = & \left\{ \langle p, p' \rangle \xrightarrow{a} \langle q, q' \rangle \;\middle|\; \begin{array}{l} p \xrightarrow{a} q \in A.\Delta \\ p' \xrightarrow{a} q' \in B.\Delta \end{array} \right\} \end{array}\right]$$

We have indeed $[\![A \oplus B]\!] = [\![A]\!] \cup [\![B]\!]$. Besides the additional subtlety of requiring completion, note how very slight changes in the construction make it serve completely different purposes.

Demo in `lecture_automata_products.py`.

10.1.2 Fully Unsynchronised Product $\|$: the Shuffle

Now let us play along a completely different axis. Instead of having a fully synchronised product, where both automata walk in complete lockstep, let us build a product where each automaton just does its thing completely independently of the other, and the product accepts once both accept.

$$A \parallel B = \left[\begin{array}{lcl} \Sigma & = & A.\Sigma \cup B.\Sigma \\ Q & \subseteq & A.Q \times B.Q \\ I & = & A.I \times B.I \\ F & = & A.F \times B.F \\ \Delta & = & \left\{ \langle p, p' \rangle \xrightarrow{a} \langle q, p' \rangle \;\middle|\; \begin{array}{l} p \xrightarrow{a} q \in A.\Delta \\ p' \in B.Q \end{array} \right\} \cup \\ & & \left\{ \langle p, p' \rangle \xrightarrow{a} \langle p, q' \rangle \;\middle|\; \begin{array}{l} p \in A.Q \\ p' \xrightarrow{a} q' \in B.\Delta \end{array} \right\} \end{array}\right]$$

To what language operator does this product correspond? $[\![A \parallel B]\!] = [\![A]\!] ? [\![B]\!]$.

Let us say that $u \in [\![A]\!]$ and $v \in [\![B]\!]$. The word $w = uv$ would be accepted by $A \parallel B$, as A can first run on u, then B can run on v; at the end, both end up in their final states, and so $A \parallel B$ accepts. But there is no order to the actions that must be taken, so that $vu \in [\![A \parallel B]\!]$ as well, running B first. Indeed, we could *alternate* A and B, each reading a letter (of u and v, respectively) in turn, accepting the interleaving $u_1 v_1 \ldots u_n v_n \ldots$. All this, and everything in between is possible.

The corresponding language operation is called the **shuffle**, or **interleaving**; we shall denote it by $\|$ as well. Let us define it on words, then lift it to languages. For all $a, b \in \Sigma$, $u, v \in \Sigma^*$, we have:

$$u \parallel \varepsilon \quad = \quad \varepsilon \parallel u \quad = \quad \{u\} \quad \text{and} \quad au \parallel bv \quad = \quad a(u \parallel bv) \cup b(au \parallel v) \;.$$

For instance, $ab \parallel cd = \{\, abcd, acbd, acdb, cabd, cadb, cdab \,\}$. For languages, we let

$$K \parallel L \quad = \quad \bigcup_{\substack{k \in K \\ l \in L}} k \parallel l \;,$$

and with this, we have indeed $[\![A \parallel B]\!] = [\![A]\!] \parallel [\![B]\!]$.

Demo in `lecture_automata_products.py`.

10.1.3 Vector-Synchronised Product

We have seen two extremes: complete synchronisation, and complete asynchronicity. Now we want to come up with a flexible notion of product that encompasses and generalises all that, and enables us to synchronise selectively on some symbols, while allowing concurrent activity of the sub-systems as desired.

The idea is to no longer force sub-automata to work upon the same alphabet, but to keep track of the symbols of each automaton separately. Our product alphabet will thus be $A.\Sigma \times B.\Sigma$, and we will choose which couples (a, b) will be synchronised, and, using a special symbol _ ("stay"), the couple $(a, \_)$ means that A may read a on its own, B remaining unchanged; likewise $(\_, b)$ means B may read b on its own, A remaining in the same state. Those tuples are called **synchronisation vectors**. A set of such vectors is called a **synchronisation set**, and is a parameter that we shall provide to our product so as to determine its behaviour.

Let us write this in all generality. Let the $A_k = \langle \Sigma_k, Q_k, I_k, F_k, \Delta_k \rangle$ be NFA, with $\_ \notin \bigcup_k \Sigma_k$. Let

$$S \quad \subseteq \quad \prod_k [\Sigma_k \sqcup \{\_\}]$$

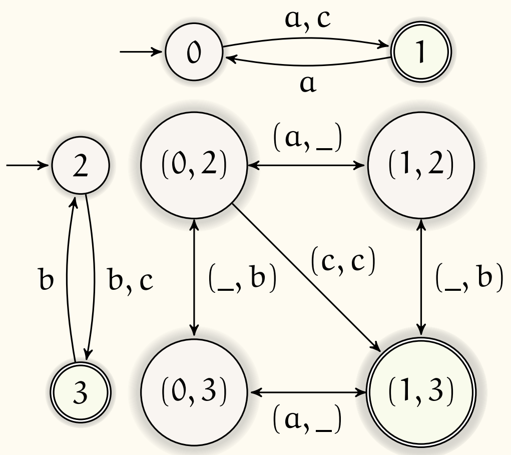
be our **synchronisation set**. Then the **vector-synchronised product of the A_k according to S** is given by

$$\bigotimes_k^S A_k \quad = \quad \left[\begin{array}{lcl} \Sigma & = & S \subseteq \prod_k [\Sigma_k \sqcup \{\_\}] \\ Q & \subseteq & \prod_k Q_k \\ I & = & \prod_k I_k \\ F & = & \prod_k F_k \\ \Delta & = & \left\{ \vec{p} \xrightarrow{\vec{v}} \vec{q} \;\middle|\; \begin{array}{l} \vec{v} \in S \\ \forall k,\ \bigvee \begin{cases} p_k \xrightarrow{v_k} q_k \in \Delta_k \\ v_k = \_ \wedge p_k = q_k \in Q_k \end{cases} \end{array} \right\} \end{array}\right]$$

Of course, I and especially F may vary according to our needs and applications; the important parts of the definition are Σ, Q, Δ.

END THIRD LECTURE (2020–2021) Will come back to that next time.

Example: Let us see a example product of two very simple automata, with synchronisation set $S = \{ (a, \_), (\_, b), (c, c) \}$:



10.1.3.1 *A Fully General Product*

Let us convince ourselves that this, composed with an homomorphism, fully generalises all previously seen notions of product.

(1) $S = \{ a^n \mid a \in \Sigma \}$: synchronises on reading the same symbol. Compose with homomorphism $h : a^n \mapsto a$ to obtain $\otimes$. Likewise for $\oplus$, with a change in final states.

(2) $S = \left\{ \vec{v} \mid \exists k,\ v_k \in \Sigma_k,\ \forall i \neq k,\ v_i = \_ \right\}$, that is to say the set of all vectors of the form $\langle \_, \_, \dots, a, \dots, \_ \rangle$: concurrent action on all symbols. Compose with

$$h : \langle \_, \_, \dots, v_k, \dots, \_ \rangle \mapsto v_k$$

to obtain $\|$.

(3) It should be clear at this point that, by playing with the synchronisation set, any combination of those behaviours can be specified.

Note: $\_^n$ is generally considered an uninteresting synchronisation vector, as it would merely create a loop upon every state of the system.

10.1.3.2 *Easy to Understand, Cumbersome to Use*

The definition above is what you will find in the literature. It is easy to write, understand, and manipulate — for a mathematician — but is not very *practical* for modelling.

Generally, you have a ton of subsystems, and only a few communicate. That means you will have to write and generate a bunch of vectors of the form

$$\langle \_, \_, \_, \_, \_, a, \_, \_, \_, b, \_, \_, \_, \_ \rangle$$

to synchronise two subsystems, and a lot of vectors of the form

$$\langle \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, x, \_ \rangle$$

to enable internal transitions of each system.

The synchronisation set is dependant upon the order of the sub-systems, which is irrelevant and arbitrary. When writing synchronisation sets, what we want to specify are things like “Farmer and Wolf cross the river left-to-right together”. Let’s say the symbol for “going left-to-right” is 1 for both systems. To specify the above, we have to find out the index of the systems Wolf and Farmer and fill everything else with _.

Assuming the order is $\langle W, \mathrm{G}, \mathrm{C}, \mathrm{F} \rangle$, the synchronisation vector is $\langle 1, \_, \_, 1 \rangle$. Even with four systems, that is not very legible. Imagine with 20:

$$\langle \_, \_, 1, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, 1, \_, \_, \_, \_, \_, \_ \rangle$$

And yet, only two systems are actually pertinent. And woe betide you should you wish to reorder the systems. Every single vector must then be reordered. This, again, despite the order being ultimately arbitrary and meaningless.

Programmatically, having to handle arbitrary indexes adds another layer of complexity and indirection to writing and interpreting specifications. I originally implemented the vector product in my NFA framework, and quickly realised that it was a pain to use, and I wasn’t having any fun modelling problems with it. So I wrote and implemented a different, in my opinion much more usable, version of the product, presented in the next section, which is what we are going to use for this course. The drawback is that it takes more time to define the data types, but since you define things once and use them many times, it remains a good investment.

Again I stress that the vector version is the “canonical” version that you will find in other lecture notes and books on the subject.

10.1.4 Named Synchronised Product

This is my personal take on the parametric synchronised product, which we shall use to model and implement systems from now on. The idea is that synchronisations of the form “Farmer and Wolf cross the river left-to-right together”, or, in terms of symbols, “Farmer and Wolf synchronise on symbol 1; other systems don’t care”, should be written in the simplest way possible, as

$$\{ \mathrm{Farmer} : 1, \ \mathrm{Wolf} : 1 \} .$$

That is to say, instead of a set of synchronisation *vectors*, we have a set of synchronisation *maps*, or, in Python’s terms, *dictionaries*[(d)], where we simply *name* the relevant systems, and ignore all the irrelevant ones, however many there may be.

[(d)] I am going to use *map* in the lecture notes, simply because it’s easier to type 50 times than *dictionary*. Also it’s more general terminology.

Let $\mathcal{C} = \{ A_1, \ldots, A_n \}$ be a collection of NFA, referred to as the *components* which we assemble into a global system. We let

$$\mathbb{D}(X) \quad = \quad \mathcal{C} \to X \quad \subseteq \quad \wp(\mathcal{C} \times X) \quad \text{and} \quad \mathbb{D}_p(X) = \mathcal{C} \nrightarrow X$$

be the type of **X-valued (total, resp. partial) component mappings**. That is to say, an element of $\mathbb{D}(X)$ is a mapping that associates an element of the set X to each component. For instance,

$$\{ A_1 : 1, \ldots, A_n : n \} \quad \in \quad \mathbb{D}(\mathbb{N}) \; .$$

However, we need more specific mapping types, that express the concept of associating to each automaton A an element of, say, $A.Q$, or $A.\Sigma$. This we would write as $\mathbb{D}(.Q)$ and $\mathbb{D}(.\Sigma)$, respectively. For instance, the synchronisation map

$$\{ \text{Farmer} : 1,\ \text{Wolf} : 1 \} \quad \in \quad \mathbb{D}_p(.\Sigma)$$

Associates to *some* (partial mapping) components (the Cabbage and the Goat are left alone) one of *their* transitions. It would be inappropriate indeed if the map associated to the Farmer a transition outside of Farmer.Σ; each component must have its own target space. Formally, we let

$$\mathbb{D}(.X) \quad = \quad \left\{ d \in \mathbb{D}\Big(\bigcup_{c \in \mathcal{C}} c.X\Big) \;\middle|\; \forall c \in \mathcal{C},\ d(c) \in c.X \right\}$$

be the set of **attributed X-valued total component mappings**. The partial version is defined similarly, in the obvious way. Let

$$S \quad \subseteq \quad \mathbb{D}_p(.\Sigma)$$

be our **synchronisation set** of **synchronisation maps**. Then the **named synchronised product of $\mathcal{C}$ according to S** is given by

$$\bigotimes^S \mathcal{C} \quad = \quad \left[\begin{array}{lcl} \Sigma & = & S \quad \subseteq \quad \mathbb{D}_p(.\Sigma) \\ Q & \subseteq & \mathbb{D}(.Q) \\ I & = & \mathbb{D}(.I) \\ F & = & \mathbb{D}(.F) \\ \Delta & = & \left\{ p \xrightarrow{d} q \;\middle|\; \begin{array}{l} d \in S,\ \forall c \in \mathcal{C}, \\ \bigvee \begin{cases} p(c) \xrightarrow{d(c)} q(c) \in c.\Delta \\ c \notin \mathrm{dom}(d),\ p(c) = q(c) \in c.Q \end{cases} \end{array} \right\} \end{array} \right]$$

10.1.5 Automaton Restriction

Sometimes, we want to check whether a system can reach an undesirable state, in which case we compute the global system and look at its reachable states, issuing a verdict of “correct” or “incorrect”.

Other times, as in WGC, we are seeking a solution to a problem given certain constraints on the states — the Wolf and the Goat cannot be together without supervision — which we already know can easily be violated. In that case the question is not whether bad states can be reached, but whether a certain goal state can be reached through a path using only good states.

To do this, we compute the global automaton, and restrict it, removing all undesirable states. Let $A \in \mathrm{NFA}$ and $P \subseteq A.Q$ a subset of states or, equivalently, a predicate on states; the **restriction of A to P** is defined as

$$A|_P \;=\; \left[\begin{array}{lcl} \Sigma & = & A.\Sigma \\ Q & = & A.Q \cap P \\ I & = & A.I \cap P \\ F & = & A.F \cap P \\ \Delta & = & A.\Delta \cap \Big(P \times \big(\Sigma \cup \{\varepsilon\}\big) \times P\Big) \end{array}\right]$$

With this, we have, at long last, every tool we need to solve our problems.

Note: The experience of previous years shows that the equivalence of predicates and sets is not crystal clear to every student, so here is a reminder. The mapping

$$b : \left| \begin{array}{ccc} (Q \to \{0,1\}) & \longrightarrow & \wp(Q) \\ P & \longmapsto & P^{-1}[1] \end{array} \right.$$

establishes a bijection between predicates on Q and subsets of Q.

10.2 Example Systems, Now With Some Products

The general method for modelling with products is as follows: your target is

$$\text{The big system} \quad = \quad \bigotimes^{S} \mathcal{C}\Big|_{\ell} \,.$$

To get there:

(1) Identify the set $\mathcal{C}$ of sub-systems, and give an automaton for each.

(2) Identify how they must synchronise with each other, and give a synchronisation set S that formalises this.

(3) *Optionally*, give a filter (set, predicate,...) ℓ to focus on *licit* states and discard other reachable states.

This last step usually intervenes mostly when you are looking for the solution to a problem under constraints (such as WGC), as opposed to modelling a system and checking whether it can end up in undesirable states. In the second case, you usually have a set B of bad states in mind, and the question is whether

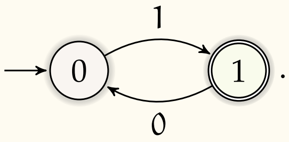
$$\Big(\bigotimes^{S} \mathcal{C}\Big).Q \cap B \;=\; \varnothing \,.$$

10.2.1 WGC, Now With Map-Synchronised Product

Let us apply this approach to WGC. (With some digressions on the way. . .)

(1) Give components $\mathcal{C}$:

Each actor can move from one bank to the other, under some synchronisation conditions. Thus, they are instances of this simple automaton:



Since the goal is to reach the right bank, we can simply define the corresponding state 1 as final, and the product will target the desired result of "everybody is on the right" without any additional fiddling with final states.

We have thus an automaton for each of Wolf, Goat, Cabbage, and Farmer:

$$\mathcal{C} = \{W, G, C, F\}$$

Again, we do not model the Boat, for the same reasons as in the naïve approach.

(2) Give synchronisation set S:

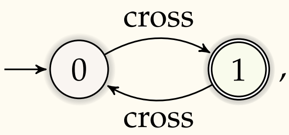
The rule for components changing states is: "the Farmer and at most one other actor on the same bank can jointly change state; nothing else".

Thus we have the synchronisation set:

$$S = \big\{ \{F : x, a : x\} \,\big|\, x \in \{0, 1\},\ a \in \mathcal{C} \big\} . \tag{10.3}$$

Note that since executing, say, the transition 1, "go to right", is only possible if you are on the left, state 0, and vice-versa, this *does* ensure that both actors must be on the same bank, though the synchronisation map has no direct access to states.

For this reason, choosing



for the components would ultimately have been a bad idea, as we would have no means to enforce that both actors are on the same bank when they cross.

The moral of this is "*if your synchronisations are dependant upon state data, then you need to encode the relevant data in your subsystems' transitions*".

Digression: For purposes of comparison, using the vector product in order $\langle W, G, C, F\rangle$, we would have had the following synchronisation set:

$$S = \Big\{\langle \_, \_, \_, 0\rangle, \langle 0, \_, \_, 0\rangle, \langle \_, 0, \_, 0\rangle, \langle \_, \_, 0, 0\rangle,\\ \langle \_, \_, \_, 1\rangle, \langle 1, \_, \_, 1\rangle, \langle \_, 1, \_, 1\rangle, \langle \_, \_, 1, 1\rangle\Big\}\ .$$

Written in extenso, we have the following synchronisation maps:

$$S = \Big\{\{F:0\},\{W:0,\ F:0\},\{G:0,\ F:0\},\{C:0,\ F:0\},\\ \{F:1\},\{W:1,\ F:1\},\{G:1,\ F:1\},\{C:1,\ F:1\}\Big\}\ .$$

As you can see, it is easier to see at a glance what they mean, and they are also easier to define in intension, as in Eq. (10.3). The closest we can come in the vector version is to define a set

$$M = \{\{F, a\} \mid a \in \mathcal{C}\}$$

and map it to a vector version, with respect to the positions in $\langle W, G, C, F\rangle$, filled with 0 (right-to-left), and another vector version with 1 (left-to-right). And this does not even generalize well to cases where actors do not all cross in the same direction.

This is why I moved away from the vector version in class. Again, you still need to *understand* it, but for purposes of problem-solving, I'll not only allow but favour the map version of the product.

(3) If relevant, give a filter ℓ:

We are solving a problem, avoiding certain states, and thus we have need of a filter. As for the naïve method we take, for any $s \subseteq \mathcal{C}$ on the same bank,

$$\ell_0(s) = \big[\{W, G\} \subseteq s \vee \{G, C\} \subseteq s\big] \implies F \in s\ ,$$

and both banks must be licit: we have

$$\ell(q) = \ell_0\big(q^{-1}[0]\big) \wedge \ell_0\big(q^{-1}[1]\big)\ ,$$

where $q^{-1}[0]$ is the *preimage of* $\{0\}$ *under* q. If that is unclear, recall that q is a mapping from components to states. The preimage of q, defined for all sets S as

$$q^{-1}[S] = \{x \mid q(x) \in S\}$$

is thus a mapping from sets of states to the set of components q sends to those states.

Here, $q^{-1}[0]$, short for $q^{-1}[\{0\}]$, is thus the set of components that are in their state 0 when the global system is in state q.

Another way of writing ℓ, without going through another function ℓ_0, would be

$$\ell(q) \;=\; \forall x \in \{0,1\},\ \big(q[W,G] = \{x\} \ \vee\ q[G,C] = \{x\}\big) \implies q(F) = x\,,$$

where $q[\cdot]$ is the image function, following the same conventions as the preimage. But that solution does not seem clearer than the previous one. Which notations work best will depend on the problem at hand.

Look how it's done in `wolf.py`.

END FOURTH LECTURE (2020–2021) Skipped digression

END THIRD LECTURE (2021–2022) Skipped digression

10.2.2 Indiana Jones, now With Map-Synchronised Product

Let us come back to Sec. 10.0.4[p46]: "Indiana Jones and the Temple of Verification", now with products. I'll first give the solution with little commentary, just as you would be supposed to provide it during an exam, for instance. Then we'll discuss the reasoning behind some of the choices here.

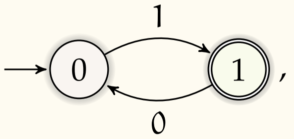
10.2.2.1 The Solution, Concisely

The added difficulty compared to WGC is the handling of the time component. Fortunately, we have already seen how to model an incrementing/decrementing variable in Sec. 8.4.4[p40]: "Incrementable Integer Variable", and so we have a Time component

$$T \;=\; V\big(0, 15, 15, \{-1,-2,-4,-8\}\big)\,.$$

All states of T are final, since we don't care how much time is left when we win.

All other components Indy, Girls, Wounded, Child, and the Lamp, are instances of the usual left/right automaton



and we have

$$\mathcal{C} \;=\; \{I, G, W, C, L, T\}\,.$$

We synchronise on

$$S \;=\; \big\{\{L:x, a:x, b:x, T:-\tau\{a,b\}\} \,\big|\, x\in\{0,1\},\ a,b \in \mathcal{C}\setminus\{L,T\}\big\}\,,$$

τ being the same time function as in the naïve method.

We have no filter on states.

END THIRD LECTURE (2022–2023) Skipped digression

10.2.2.2 *How Was I Supposed to Guess How to Handle Time?*

From discussions in lab classes, it seems it is not immediately obvious that time should be handled as a component, as an automaton, and then synchronised. How could you guess that? Let us take a look at the state space of the naïve version in Sec. 10.0.4[p46]: "Indiana Jones and the Temple of Verification", Eq. (10.1):

$$Q \quad \subseteq \quad \wp(A) \times [\![0, 15]\!]$$

Recalling that $\wp(A)$ is equivalent to a function $\{\,0, 1\,\}^{A}$ or, given an ordering an A, a vector $\{\,0, 1\,\}^{|A|}$, a state was equivalent to a vector

$$\langle \underbrace{x_I, x_G, x_W, x_C, x_L}_{0/1 \text{ side info}}, \underbrace{t}_{\text{Time}} \rangle \ ,$$

and the rule of thumb is that if your target state space in the naïve model boils down to a product

$$\prod_{k=1}^{n} Q_k \ ,$$

even if it is perhaps not written exactly that way, then it is probably a good idea to have a product model with n components whose state spaces are the Q_k. Which should not be surprising since the product of those components will have state space $\prod_{k=1}^{n} Q_k$ — or the equivalent in terms of dictionaries — by definition.

It *would* be extremely surprising if the product model and the naïve model had fundamentally different state spaces; after all, they model the *same* problem, albeit with different notations and approaches. If the two approaches yielded fundamentally different results, it would be a safe bet that at least one of them is flawed.

Given a problem or system to model, all approaches should generate isomorphic models. That is to say, they may differ in how the information within their states and transitions is encoded, the "names" of their states and transition labels, but at the end of the day this is arbitrary for our purposes; what fundamentally matters is the state/transition *structure*, the shape of the graph.

10.2.3 Exercise with Solution: The Bridge on the River Kwaï (FR)

This was an exercise in the 2020 final exam. Use it to train yourself, without looking at the solution below prematurely.

I advise writing all your answers down first, in 40min to an hour maximum, and then putting your pen down and comparing with the solution.

10.2.3.1 Problem Statement

Le Capitaine et un Soldat veulent traverser la rivière Kwaï. Malheureusement, quelqu'un a fait sauter le pont, et ils ne savent pas nager.

Le Capitaine avise deux enfants s'apprêtant à embarquer dans un frêle esquif. Il s'avère que leur embarcation est trop fragile pour supporter davantage que le poids d'un militaire seul, ou le poids total des deux enfants, Alice et Bob.

Après négociations, les enfants acceptent d'aider les militaires. Cependant, Alice est intimidée par la moustache du Capitaine, et refuse de rester seule avec lui. Tout le monde commence sur la rive gauche, et tous peuvent conduire la barque.

On veut savoir si, et comment, les militaires peuvent traverser la rivière avec ces ressources; si tout le monde peut traverser; et autres question similaires.

Dans cet examen, on ne demande pas de *résoudre* ces problèmes, mais de modéliser la situation formellement grâce à un produit synchronisé d'automates (version dictionnaires) $\bigotimes^S \mathcal{C}\Big|_\ell$.

(1) Donner l'ensemble $\mathcal{C}$ des composants / sous-systèmes. On ne se préoccupera pas de leurs états finaux.

(2) Donner l'ensemble S des dictionnaires de synchronisation.

(3) Donner si utile un filtre ℓ sur les états du système synchronisé $\bigotimes^S \mathcal{C}$. [e]

Le scénario se reproduit le lendemain. Cette fois, les enfants décident d'embrasser le capitalisme. Chaque fois qu'un enfant traverse la rivière, seul ou non, il facture 1§ (un Simflouz) aux militaires, et un militaire doit payer 4§ pour chaque utilisation de la barque.

Sachant que les militaires ont un budget initial de $N = 18$§, on veut savoir combien traverser la rivière leur coûtera, et autres questions similaires. On adapte la modélisation.

(4) $\mathcal{C}$ change-t-il ? Si oui comment ?

(5) Même question pour S. Le redéfinir entièrement s'il y a un changement.

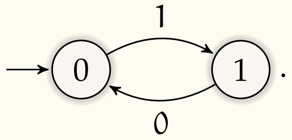
(6) Même question pour ℓ.

[e] Aide aux notations: On rappelle qu'un état q d'un produit $\bigotimes^S \mathcal{C}$ est un dictionnaire (i.e. une fonction) de type $q : \mathcal{C} \to \bigcup_{c \in \mathcal{C}} c.Q$ associant à chaque composant un de ses états.

On ne se privera pas d'utiliser l'image inverse q^{-1} d'un état si c'est pratique pour définir ℓ. Exemple Loup/Chèvre/Chou: $q^{-1}[0]$ est l'ensemble des composants qui sont dans l'état 0, i.e. la rive gauche.

10.2.3.2 *Solution*

(1) On a le système habituel rive gauche/droite



pour chacun des acteurs Alice, Bob, Capitaine, Soldat, Esquif. (J'évite "Barque" parce que ça ferait 2 B).

$$\mathcal{C} \;=\; \{\, A, B, C, S, E \,\} \,.$$

Ce coup-ci on modélise bien l'Esquif, car il n'a pas le même comportement qu'un autre acteur, contrairement à WGC, où la Barque a le même comportement que le Fermier.

Dans les copies j'ai eu pas mal de monde qui a fait l'impasse sur l'Esquif, ayant apparemment retenu du cours sur WGC qu'"on ne modélise pas les bateaux". Ce n'est pas la bonne leçon à retenir...

(2) On traduit assez directement les mouvements possibles:

$$\begin{aligned} S \;=\; & \Big\{ \{\, E : x, c : x, d : x \,\} \;\Big|\; x \in \{\, 0, 1 \,\},\; c, d \in \{\, A, B \,\} \Big\} \\ & \qquad\qquad\qquad \cup \Big\{ \{\, E : x, c : x \,\} \;\Big|\; x \in \{\, 0, 1 \,\},\; c \in \{\, C, S \,\} \Big\} \end{aligned}$$

(3) Alice et le Capitaine ne doivent pas être seuls sur un rive. Notons que l'esquif ne compte pas comme une présence rassurante.

Soit $r \subseteq \mathcal{C}$ le "contenu" d'une rive:

$$\ell'(r) \quad = \quad \{\, A, C \,\} \subseteq r \implies (B \in r \vee S \in r)$$

Les contenus des deux rives doivent être sans conflits:

$$\ell(q) \quad = \quad \ell'\big(q^{-1}[0]\big) \;\wedge\; \ell'\big(q^{-1}[1]\big)$$

(4) Oui, on doit ajouter un acteur représentant le porte-monnaie des militaires.

On peut facturer 1§ pour un enfant seul, 2§ si les deux traversent en même temps (1§ par enfant), ou 4§ pour un militaire.

On a donc:

$$P \;=\; V(0, N, 0, \{\, 1, 2, 4 \,\})$$

$$\mathcal{C}' \;=\; \mathcal{C} \cup \{P\}$$

Par pitié, en examen, ne pas déplier la définition de $V(0, N, 0, \{1, 2, 4\})$! Quasiment tout le monde a fait ça dans les copies, dans un examen où la question précédente demandait de définir V. Je ne comprends pas le raisonnement. En programmation, quand on vous demande de réutiliser une fonction, vous recopiez le code au lieu d'appeler la fonction ? Bien sûr que non. C'est précisément pour ne pas recopier cent fois le même code qu'on a inventé les fonctions. Même combat ici.

(5) Il faut juste ajouter la facturation:

$$\begin{aligned} S' \;=\; & \big\{\{E : x, c : x, d : x, P : |\{c, d\}|\} \;\big|\; x \in \{0, 1\},\; c, d \in \{A, B\}\big\} \\ & \qquad \cup \big\{\{E : x, c : x, P : 4\} \;\big|\; x \in \{0, 1\},\; c \in \{C, S\}\big\} \end{aligned}$$

(6) Rien à changer. On n'ajoute pas de contrainte sur les états, à part "il faut avoir du pognon", mais ça c'est déjà couvert par l'espace d'états de P.

END FOURTH LECTURE (2021–2022)

10.2.4 Exercise with Solution: The Toggle Problems

This was an exercise in the 2022 final exam (FISE).

There is a kind of puzzle often found in video games[f]. I call them "toggle puzzles".

10.2.4.1 Problem Statement

Let us begin with a small-ish, specific instance:

> *You find four switches (e.g. light switches, levers, or anything else with two states that can be flipped or toggled), initially all "off", and the goal is to put them all in the "on" position.*
>
> *The problem is, the switches are linked, so that manually flipping one also flips others at the same time:*
>
> - *manually flipping the first switch also automatically flips the third,*
> - *manually flipping the second switch also flips the first,*
> - *manually flipping the third switch also flips the second and fourth,*
> - *manually flipping the fourth switch also flips the first.*

[f] There is a very difficult instance in the tutorial level of *Pathfinder: Kingmaker (2018)* (6 switches, 64 states!), and a small one in *Vampire: The Masquerade – Bloodlines (2004)* — with a random twist. A colleague reminds me that there is one as well near the beginning of Skyrim (2011); this one is rather trivial, if I recall. I'd love more examples if you have them.

We'll solve them via state space analysis, but I'm sure there are more scalable linear algebra solutions, and I'd love to see a writeup on that.

Note that the rules are independent; for instance, *manually* flipping the first causes the third to be *automatically* flipped, but that does *not* mean that the second and fourth must also be flipped.

You will model this situation formally as a map-synchronised product

$$T = \bigotimes^{S} \mathcal{C}\Big|_{\ell} \;.$$

The final states of T must be such that all switches are "on".

(1) Give the set $\mathcal{C}$ of components / sub-systems.

(2) Give the set S of synchronisation maps.

(3) Give, if useful, a filter ℓ upon the states of the synchronised system $\bigotimes^{S} \mathcal{C}$.

Now let us generalise to all possible problems of this form:

> *You find* N *switches* $s_1, \dots, s_N$*, initially all off, and the goal is to turn them all on.*
>
> *They are linked by a "flipping function"*
>
> $$f \in \{ s_1, \dots, s_N \} \to \wp(\{ s_1, \dots, s_N \})$$
>
> *so that flipping a switch* s_k *also flips all switches in* $f(s_k)$ *at the same time.*

Again, let us model this using synchronised products.

(4) Give the set $\mathcal{C}$ of components / sub-systems.

(5) Give the set S of synchronisation maps.

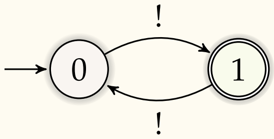
(6) Give, if useful, a filter ℓ upon the states of the synchronised system $\bigotimes^{S} \mathcal{C}$.

10.2.4.2 Solution

(1) *Give the set* $\mathcal{C}$ *of components / sub-systems.*

There are four switches; let us call them s_1, s_2, s_3, s_4. Each has two states: on and off, or 0 and 1. Unlike WGC, we explicitly do *not* want to care about the current state of the switch when flipping it, so we use a single transition symbol:

$s_1, s_2, s_3, s_4 \quad =$



There is nothing else at play here, so we take

$$\mathcal{C} = \{ s_1, s_2, s_3, s_4 \}$$

(2) *Give the set S of synchronisation maps.*

It suffices to synchronise the switches in groups following the specification of the problem. If flipping s_1 flips s_3 at the same time, it simply means that we can flip s_1 and s_3 together, etcetera:

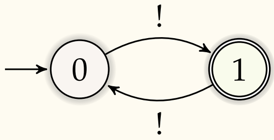
$$S = \{\{s_1 : !,\ s_3 : !\}, \{s_2 : !,\ s_1 : !\}, \{s_3 : !,\ s_2 : !,\ s_4 : !\}, \{s_4 : !,\ s_1 : !\}\}$$

(3) *Give, if useful, a filter ℓ upon the states of the synchronised system $\bigotimes^S \mathcal{C}$.*

There is no need for that.

(4) *Give the set $\mathcal{C}$ of components / sub-systems.*

$s_1, \dots, s_N \quad =$



We take $\mathcal{C} = \{s_1, \dots, s_N\}$.

(5) *Give the set S of synchronisation maps.*

Each association $f(s_k) = \{s_{k1}, \dots, s_{kn}\}$ gives us a group $\{s_k, s_{k1}, \dots, s_{kn}\} = \{s_k\} \cup f(s_k)$ of switches to synchronise:

$$S = \left\{ \{ s : ! \mid s \in \{s_k\} \cup f(s_k) \} \;\middle|\; k \in [\![1, N]\!] \right\}.$$

(6) *Give, if useful, a filter ℓ upon the states of the synchronised system $\bigotimes^S \mathcal{C}$.*

There is still no need for that.

10.2.5 Exercise with Solution: The Three Islands, the Two Wolves, the Goat, and the Cabbage

This was an exercise in the 2021 final exam.

10.2.5.1 *Problem Statement*

This exercise is your *favourite* problem, but with twice as many wolves, and instead of two river banks, three islands. More is better.

> *Once upon a time, there were three magical islands; Market Island, That Other Island, and Farmer Island, on which lived a kindly farmer.*
>
> *One day, the farmer got in his little boat, went to Market Island, and purchased two wolves, a goat, and a cabbage. The farmer's boat could carry only himself and a single one of his purchases: a wolf, the goat, or the cabbage. Every island is accessible from the others.*

If left unattended together on any island, a wolf would eat the goat, or the goat would eat the cabbage.

How could the farmer possibly get back home with all his purchases intact?

You will model this situation formally as a map-synchronised product

$$P = \bigotimes^{S} \mathcal{C} \Big|_{\ell} \, .$$

The final states of P must be such that everyone is safely on Farmer Island.

(1) Give the set $\mathcal{C}$ of components / sub-systems.

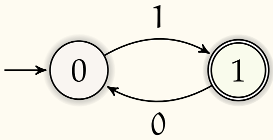
Tip: Think carefully. Actors must only be able to go together to island X if they are *both* on the same island Y. Make sure your model has enough information to enforce that.

(2) Give the set S of synchronisation maps.

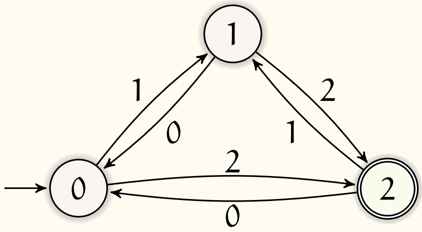
(3) Give, if useful, a filter ℓ upon the states of the synchronised system $\bigotimes^{S} \mathcal{C}$.

10.2.5.2 Solution

(1) We have three islands instead of two banks. The first instinct would be to generalise our usual two-banks system

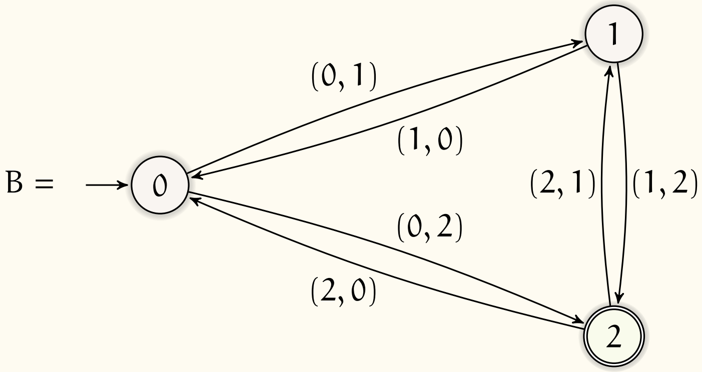


to the obvious three-banks / three-islands counterpart:

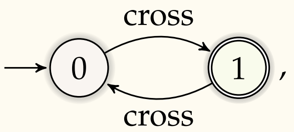


However, as famously remarked by Admiral Ackbar (back when Star Wars was cool), ***this is a trap!*** Why? Because synchronising on 2 would enable an actor on 0 and an actor on 1 to both teleport to 2. This is illegal! They can only move together if they are on the same island, as there is only one boat.

In the case of the two banks, there was no ambiguity. If you wanted to go on 1, you could only be on 0, and vice-versa. Here, we need to specify our starting point explicitly, along with our destination. Thus, we have the following basic system:



This is an extension of the principle already discussed when we saw why, for the classical WGC problem, we could not do with



which did not give us enough information to determine on which bank we were.

We reiterate the moral seen then: *"if your synchronisations are dependant upon state data, then you need to encode the relevant data in your subsystems' transitions"*.

Note: many student solutions numbered the transitions as $[\![0, 5]\!]$. *This is fine mathematically, but a bit strange to me, as it is not the natural, intuitive solution. Did they not understand that* $0, 1$ *in WGC were the target banks? Or was it to save time writing? Did that idea circulate through Discord or other channels?*

We have the components

$$\mathcal{C} \;=\; \{\, W_1, W_2, G, C, F \,\} \,,$$

all of which are copies of the basic system B.

As for classical WGC, we do not model the boat, as it is tied to the farmer anyway.

(2) We have the synchronisation maps

$$S \;=\; \left\{ \{\, F : x, a : x \,\} \;\middle|\; x \in B.\Sigma,\; a \in \mathcal{C} \right\} .$$

The only change from WGC is that the transitions are not only in $\{\, 0, 1 \,\}$ but in $B.\Sigma$. There is no need to give $B.\Sigma$ explicitly; it is already given by definition of B.

Another way of writing it, making $B.\Sigma$ explicit:

$$S \;=\; \left\{ \{\, F : (x, y), a : (x, y) \,\} \;\middle|\; x, y \in \{\, 0, 1, 2 \,\},\; x \neq y,\; a \in \mathcal{C} \right\} .$$

(3) The filter ℓ is a variant of that of classical WGC, accounting for the two wolves:

$$\ell_0(s) \quad = \quad \big[\{\,W_1, G\,\} \subseteq s \;\vee\; \{\,W_2, G\,\} \subseteq s \;\vee\; \{\,G, C\,\} \subseteq s\big] \implies F \in s\,,$$

and all islands must be licit:

$$\ell(q) \quad = \quad \forall i \in [\![0, 2]\!],\; \ell_0\big(q^{-1}[i]\big)\,.$$

10.2.6 Exercise with Solution: Max-Weight River-Crossing Problem

This was an exercise in the 2021 final exam.

10.2.6.1 *Problem Statement*

In this exercise, we generalise *The Worm, the Centipede, and the Grasshopper* to an arbitrary number of actors of arbitrary weights.

As usual, we have a river, separating two banks, and a boat, initially moored near the left bank, as the only means of crossing it.

Let $A = \{\,a_1, \ldots, a_n\,\}$ be a set of n "actors" starting on the left bank. They all wish to cross to the right bank. All have positive weights, given by the weight function $w : A \to \mathbb{N} \setminus \{0\}$,

The boat cannot operate itself, but any actor can operate it to cross the river, in either direction. The boat is quite spacious: there is no limit upon the number of actors to be ferried simultaneously. However, the boat is quite shallow and flimsy, and therefore, it cannot transport more than $M \in \mathbb{N}$ units of weight without risking to capsize.

You will model this situation formally as a map-synchronised product

$$P = \bigotimes^{S} \mathcal{C}\Bigg|_{\ell}\,.$$

The final states of P must be such that every actor is on the right bank.

(1) Give the set $\mathcal{C}$ of components / sub-systems.

(2) Give a predicate $f : \wp(A) \to \mathbb{B}$ such that $f(X)$ is true iff the boat still floats when all the actors of $X \subseteq A$ are onboard.

(3) Give the set S of synchronisation maps. *Tip: use* f.

(4) Give, if useful, a filter ℓ upon the states of the synchronised system $\bigotimes^S \mathcal{C}$.

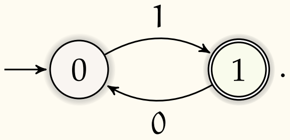
10.2.6.2 *Solution*

(1) $\mathcal{C} = A \cup \{B\} = \{\,a_1, \ldots, a_n, B\,\}$, the latter being the boat.

Aside: I have seen so many papers writing $\mathcal{C} = \{\,A, B\,\}$*, with one even explicitly writing* $\{\,A, B\,\} = \{\,a_1, \ldots, a_n, B\,\}$*. This is of course terribly wrong as a matter of basic set theory* *and immediately forfeits any and all points to the question, even if you really* meant $A \cup \{B\}$.

I have also seen $A \cup B$*, which is wrong as well, of course, but much more excusable as a typo; that cost only a few points.*

All are modelled as the usual system



The right bank is final, which ensures that the final state of the product satisfies "everybody on the right bank".

(2) The boat floats if the total weight involved is at most M:

$$f(X) \quad \equiv \quad \left[\sum_{a \in X} w(a) \right] \leqslant M \, .$$

(3) The boat and any non-empty collection of actors whose total weight does not exceed the limit move together from bank to bank:

$$S \quad = \quad \left\{ \{B : x\} \cup \{ a : x \mid a \in X \} \;\middle|\; x \in \{0, 1\},\ \emptyset \neq X \subseteq A,\ f(X) \right\}$$

Aside: following the same remark as above, $\{ B : x,\ X : x \}$ *is very wrong.*

(4) There is no need for a filter upon states.

END FOURTH LECTURE (2022–2023)

10.2.7 Semaphores: first contact

After this lecture, you can do Exercise (16)[p25]

That is quite enough with funny river-crossing problems. Let us move on to more computery stuff. As announced from the beginning, we are especially interested in concurrent systems. Let us begin with a mainstay of concurrency: semaphores.

My (short) definition:

> *A* ***semaphore*** *is a variable counting the availability of a shared resource (in the simplest and most common case, 1 for "available", and 0 for "already taken").*
>
> *It is usually associated with two operations, P* (take/request)*, and V* (release)*, which processes can call. A call to P waits until the resource is made available, then takes it, and a call to V releases the resource.*

Schematically, this can be represented by the following pseudocode:

```
def semaphore sem:
  sem = 10   # initialise: e.g. 10 instances of resource
  def P(sem): wait until atomic{ if sem > 0:  sem--; break }
  def V(sem): atomic{ sem++ }
```

Tip: the P/V historical terminology comes from Dutch and is unclear (even among the Dutch, there seems to be some speculation as to their origin); in French, I have adopted the following mnemonic:

take / request	P	"**p**rendre"
release	V	"**v**aquer"

The following is a more complete definition mostly or wholly taken from Wikipedia:

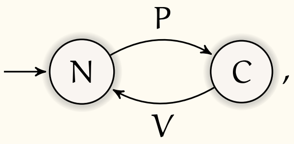
> *A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes and avoid critical section problems in a concurrent system such as a multitasking operating system. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions.*
>
> *A useful way to think of a semaphore as used in a real-world system is as a record of how many units of a particular resource are available, coupled with operations to adjust that record safely (i.e., to avoid race conditions (*concurrence critique*)) as units are acquired or become free, and, if necessary, wait until a unit of the resource becomes available.*
>
> *[...]*
>
> *To avoid starvation (*famine*), a semaphore has an associated queue (*file*) of processes (usually with FIFO semantics). If a process performs a P operation on a semaphore that has the value zero, the process is added to the semaphore's queue and its execution is suspended. When another process increments the semaphore by performing a V operation, and there are processes in the queue, one of them is removed from the queue and resumes execution. When processes have different priorities the queue may be ordered by priority, so that the highest priority process is taken from the queue first.*

Here is a pseudocode "implementation" of a trivial semaphor guarding a resource, and two processes incessantly requesting the resource, doing something with it, then freeing it:

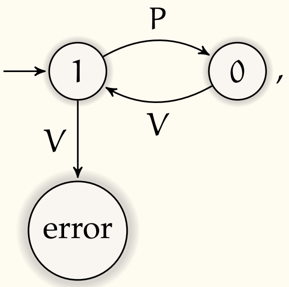
```
def semaphore sem:
  sem = 1   # initialise: one instance of resource
  def P(sem): wait until atomic{ if sem > 0:  sem--; break }
  def V(sem): atomic{ sem++ }

def process P0:
  while True:
    # noncritical section
    P(sem)
    # critical section
    V(sem)

def process P1:
  while True:
    # noncritical section
    P(sem)
    # critical section
    V(sem)

exec P0, P1
```

Let us represent this as an automaton, with N for “noncritical section” and C for “critical section” for each of the two processes, and 0, 1 for the semaphore. We have a total of $2^3 = 8$ possible states. Are all of them desirable? Probably not; we should avoid having the two processes in critical section at the same time, that’s the whole point. Are they all reachable? Are undesirable states reachable? Can the processes in danger of deadlock or starvation? Let’s find out.

The global system is a simple product of two instances of

$P_0, P_1 \quad =$  ,

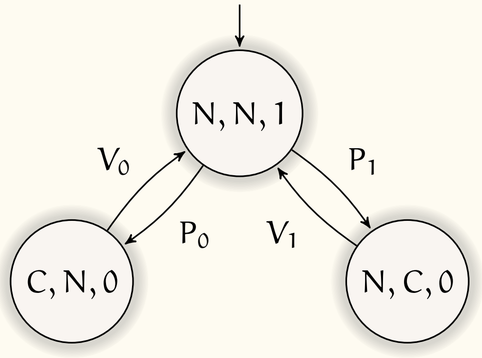
for the processes P_0 and P_1, and an instance of

$\textsc{Sem} \quad =$  ,

for the semaphore. The error state models the fact the semaphore will refuse to pretend to have more of the resource than actually available, if a process should “liberate” it without having taken it first. We synchronise on

$$S \quad = \quad \big\{ \{ \textsc{Sem} : x, P_i : x \} \,\big|\, x \in \{P, V\},\ i \in \{0, 1\} \big\}\ ,$$

and obtain the product (written in a compact way for clarity):



We see that race conditions are, fortunately, avoided, thanks to the semaphore. (The P_i in the transitions represent here the moment when the resource is actually taken, not necessarily the moment when it was *requested* via P)

Is there a guarantee of neither process starving? No. Nothing prevents the resource from being granted to P_0 again and again, while P_1 starves.

We shall see later a stronger solution, Peterson's algorithm, that guarantees starvation-freeness, and even bounded waiting.

The automata above were made intuitively. Of course we want to go towards a more general and systematic way to model such concurrent programs. . .

10.2.8 Mutable Boolean variable

Let's continue our steps into modeling concurrent systems, programs, protocols. . .

Variables are seen as atomic subsystems, storing a value in their state, and the rest of the system will request value changes and checks through synchronised messages. That includes FIFO, LIFO, and incrementing/decrementing variables seen earlier. Let us note that so far, interactions between those variables and the other components of the system were limited to *writing* operations.

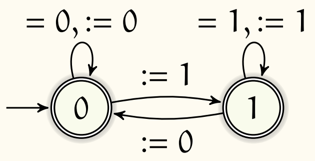
Let's deal with Boolean variables, and this time, let us model both *read* and *write* interactions with other components. The same approach can of course be applied to the other types of variables seen before to add support for read interactions.

A Boolean variable b can have states 0 (False) and 1 (True), and the possible operations are:

(1) Tests: $b = 0$, $b = 1$

(2) Assignments: $b := 0$, $b := 1$

Such a variable is thus represented by the system:



With my framework, you can use the following code to implement binary / Boolean variables:

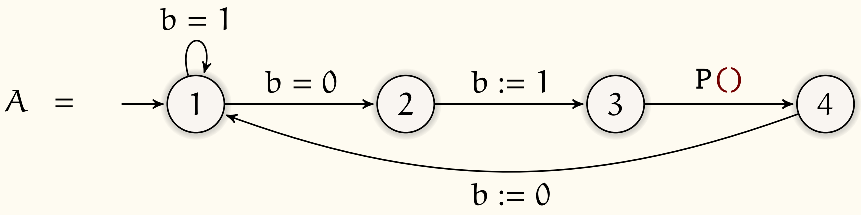
```
BinVar = NFA.spec("""
0
0 1
0 :=1 1 =0 0 :=0 0
1 :=0 0 =1 1 :=1 1
""", "BinVar").visu()

SomeVar = BinVar.copy().named("SomeVar")
OtherVar = BinVar.copy().named("OtherVar")
```

10.2.9 Sequential Programs

Consider the reactive program

```
while True:      # l1
  if not b:
    b := True    # l2
    P()          # l3
    b := False   # l4
```

Seeing tests and affectations as messages / symbols, let us model the control flow of this program, `True` and `False` being assimilated to 1 and 0 — as is the case in Python behind the scenes.



Now imagine that A and b are two concurrent systems, and that b reacts to the tests and assignment messages that A sends to it, changing its value (i.e. its *state*) and answering accordingly. `P()` can also be another system. This can be achieved by a synchronised product between the control flow automaton and the variables.

Then we would have modelled pretty much all the relevant behaviours of the global program. This is the kind of approach we are getting to in this class.

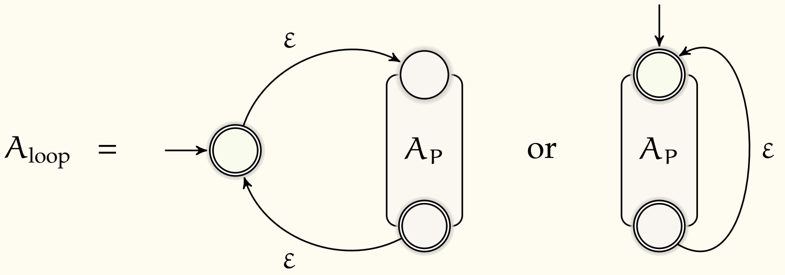
For now, this control flow automaton is obtained through intuition. Is there a systematic way to construct it? Let us examine the principal syntactic constructions and see how we can, inductively, "compile" an Abstract Syntax Tree (AST) into a control-flow automaton.

10.2.9.1 *Infinite Loop:* `while True ...`

```
while True:
  P()
```

Let, inductively, A_P be the automaton corresponding to the procedure P, and let us assume that it has exactly one initial state and exactly one final state, not so much for the recognised language, but for the purpose of using sub-automata as building blocks for larger ones.

We build the control flow automaton for the loop by repeating P indefinitely:



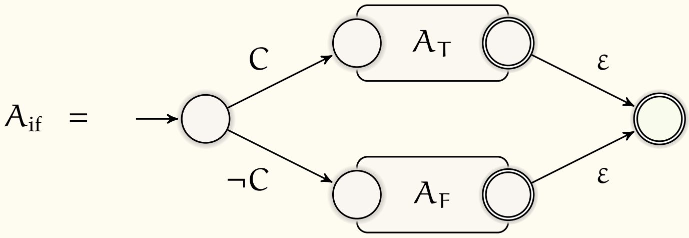
Note that the colour of the accepting state of the sub-automaton is slightly different, to indicate that it is not actually a final state of A_{loop}, just the "output point" of the subautomaton.

We shall simplify out all ε-transitions at the end.

Now let us do the same with other patterns.

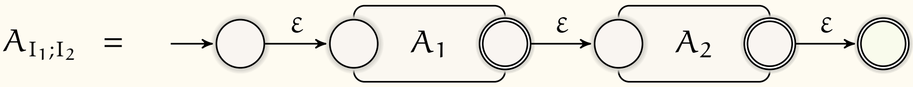
10.2.9.2 `if ... else ...`

```
if C:
  T()
else:
  F()
```



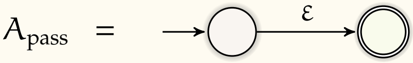
10.2.9.3 *Sequence of instructions*

```
I1
I2
```



10.2.9.4 Null operation: pass

```
pass
```

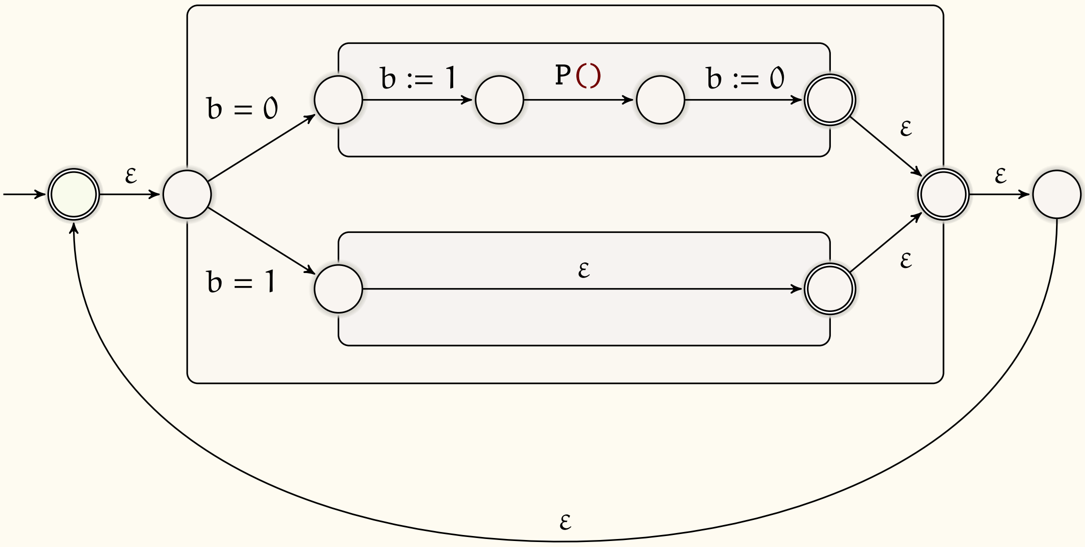


10.2.9.5 Application to our example

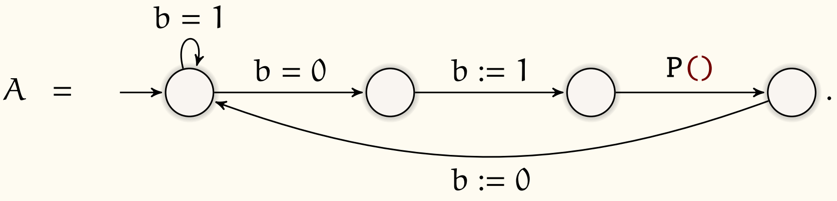
While there are of course other patterns, we now have enough to combine all those translation rules and apply them to our original reactive program example, which we rewrite a little bit:

```
while True:      # l1
  if not b:
    b := True    # l2
    P()          # l3
    b := False   # l4
  else:
    pass
```

We obtain the following system:



(I added the right-most state purely for aesthetic reasons.) Of course, this can be simplified by removing the ε-transitions, obtaining our initial automaton



Here, the procedure `P()` is abstracted, but in a real case we would need to recursively translate `P()` as well.

10.2.10 Peterson's Algorithm

Let us apply this method from beginning to end to a real-world algorithm: Peterson's method.

Here is a short description of this nice algorithm, taken from Wikipedia:

> *Peterson's algorithm (or Peterson's solution) is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication, provided changes to binary variables (bits) propagate immediately and atomically.*
>
> *It was formulated by Gary L. Peterson in 1981. While Peterson's original formulation worked with only two processes, the algorithm can be generalized for more than two.*[g]

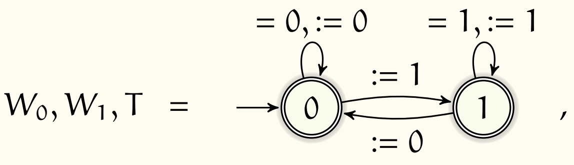
In pseudocode, the algorithm looks like this:

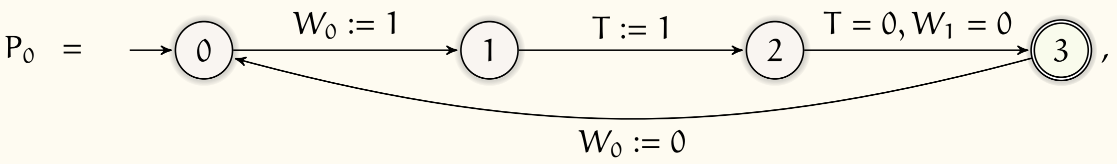
```
def binary_vars:
  W0    := 0 # process 0 wants critical access
  W1    := 0 # process 1 wants critical access
  Turn  := 0 # Whose turn is this ?

def process P0:
  while True:
    # noncritical section
    W0    := 1
    Turn  := 1
    wait until W1 = 0 or Turn = 0
    # critical section
    W0    := 0

def process P1:
  while True:
    # noncritical section
```

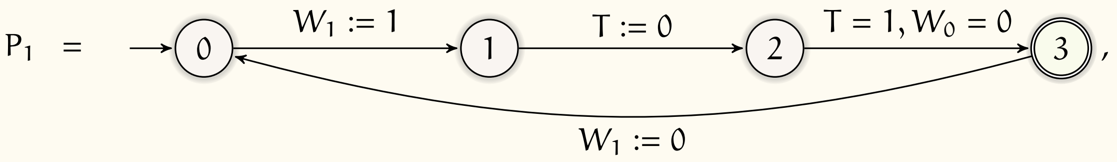
[g] Although I should add that the generalised version has weaker properties: it does not guarantee bounded waiting.

```
    W1     := 1
    Turn   := 0
    wait until W0 = 0 or Turn = 1
    # critical section
    W1     := 0

exec P0, P1
```

Modelling into automata, we have the following components (with somewhat arbitrary final states, as there is no specific goal we are trying to reach; here I highlighted states where some process is in its critical section):







which we synchronise according to the basic variable interactions:

$$\left\{ \{p : x \sim v,\ x :\sim v\} \;\middle|\; p \in \{P_0, P_1\},\ x \in \{W_0, W_1, T\},\ \sim \in \{=, :=\},\ v \in \{0, 1\} \right\}.$$

We obtain a fairly non-trivial global system.

Note that every single step we took here to obtain that model could be automated.

Demo in `Peter.py` — which I do not share; you have to implement this for yourself.

Let us check that this algorithm has all the good properties we are hoping for.

(1) Mutual exclusion.

P_0 and P_1 must not be in the critical section at the same time.

This can be verified, *proven*, to be true for our model, by checking that there is no reachable state where both processes are in critical section (state 3).

This was already achieved by a simple semaphore.

(2) **No deadlock.**

There is always a way forward, the system never locks up and stops. Some process will eventually be granted access.

The first part can be verified by the fact that every state has a transition towards another state. The second is a bit trickier, but can be seen by following the arrows.

There was no deadlock either with the semaphore.

(3) **No starvation.**

No process may be perpetually denied access to the resource. Put another way, whenever a process requests access, access will eventually be granted to that process.

This was *not* achieved by a semaphor; nothing was preventing a bad scheduling algorithm from always giving the resource to the same process in perpetuity, starving the other.

This *is* achieved by Peterson's solution. We can see this by following the arrows; in fact we have an even stronger property.

(4) **Bounded waiting / bounded bypass.**

There is a fixed number n such that no process shall ever pass their turn more than n times. That is to say, no more than n other processes will be granted access before it.

In the case of our Peterson algorithm, no process will wait more than one turn.

We can see this by following the arrows. Whenever a system requests access, it is either granted immediately, or on the next loop.

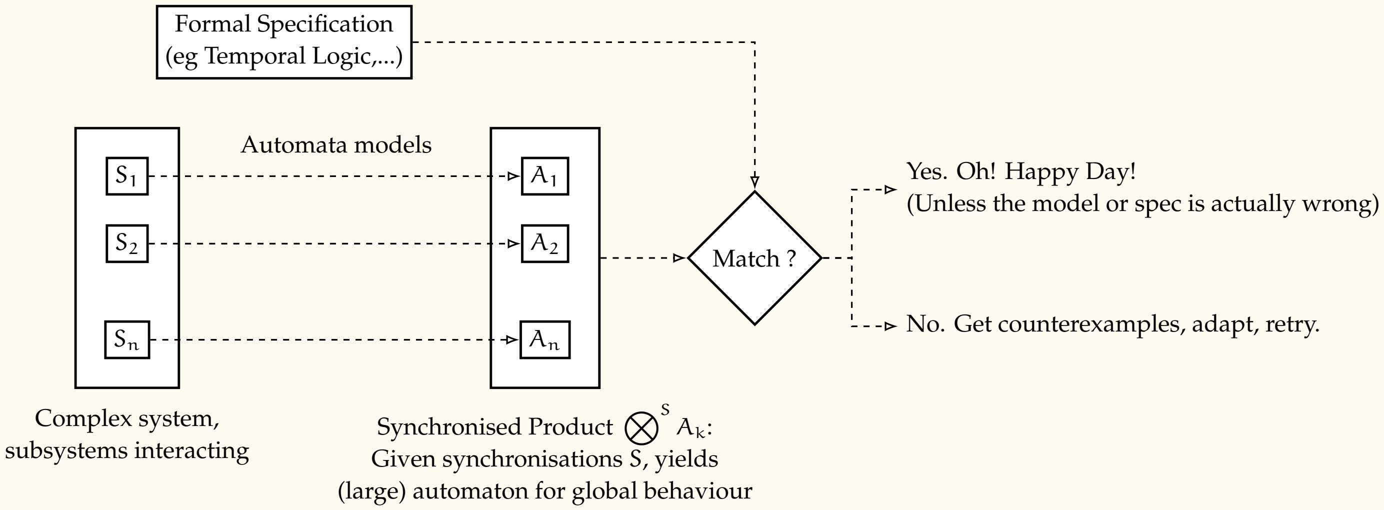
This is all very nice. So far we have managed to obtain a rigorous and systematic method to formally model complex concurrent systems.

However, except in the simplest cases and safety or security properties that reduce to the accessibility of some states, we still rely on intuition and "following the arrows" to check whether the system satisfies desired properties. Liveness properties in particular.

It would be nice if we *also* had a rigorous and systematic, formal way to express all those properties and obtain a proof of the system's compliance. Bonus points if this can be automated to at least some extent.

This is the object of the next section.

11 Classical and Temporal Logics: (W)S1S, CTL*, CTL, LTL


Formal Specification
(eg Temporal Logic,...)
Automata models
S_1
S_2
S_n
A_1
A_2
A_n
Match ?
Yes. Oh! Happy Day!
(Unless the model or spec is actually wrong)
No. Get counterexamples, adapt, retry.
Complex system,
subsystems interacting
Synchronised Product $\bigotimes^{S} A_k$:
Given synchronisations S, yields
(large) automaton for global behaviour

We need a rigorous, systematic, formal language in which to specify properties of a system's execution, which we can check against the formal model of the system, thus obtaining a proof of the system's compliance / correctness — hopefully — or a proof of non-compliance. Bonus points if, in the later case, we can obtain counter-examples which elucidate the differences between specification and system. Extra bonus points if all this can be automated to at least some extent.

Thus, not only do we want to specify the properties, we want this specification to be "instrumentable", translatable into algorithms that play well with the automata representing the system.

It so happens that we already know formal systems that satisfy those requirements. A system's execution is, *in fine*, a word whose letters are the successive states of the system, or at least some relevant properties of the states of the system. Since we are dealing with reactive systems, those words may be infinite; and there may be infinitely many possible executions.

And we *do* know how to handle possibly infinite sets of words: that is the object of Languages and Automata Theory.[h] If our properties describe regular languages, then they can be implemented as automata, and we have a clear path towards verifying our system. For instance, if we have an automaton A_S for the system S, recognising the *execution traces* of the system, and another automaton A_P recognising all possible traces satisfying a property P, then the question "is system S correct wrt. property P" translates into

$$[\![A_S]\!] \subseteq [\![A_P]\!] \text{ ?}$$

[h] We haven't seen automata on *infinite words*, only automata on finite words, of which there may be infinitely many. But there are automata on infinite words, and the theory is very similar.

Two questions remain:

(1) Which of the many regular or sub-regular formalisms shall we use to specify our properties? It is convenient, easy to write in, easily understandable once written, etc? This is important because the more impenetrable the specification language is, the less accessible verification becomes outside of very academic settings.

(2) What is the algorithmic complexity of our verification chain, for our chosen language? Systems can grow large enough on their own; if the specifications require immense automata on top of that, thing won't go well. . .

Our final choice will have to be a compromise between the expressiveness and elegance of our specification language, and its algorithmic simplicity.

In practice, following the historical evolution during the infancy of formal verification, we shall start with the more classical logics, see that they are a bit too complicated along the two axes above, and then introduce specialised modal logics, weaker but more suited to the specific task at hand and gifted with much more favourable algorithmics.

11.1 Traditional Logic on Words: (w)S1S

From a language-theoretical viewpoint, automata are a very canonical model, chock-full of desirable closure properties and equivalent in power to a slew of other formalisms:

- regular expressions,
- $\mathrm{DSPACE}(O(1))$ & $\mathrm{NSPACE}(O(1))$
- prefix grammars ($x \twoheadrightarrow y$ if $x = vu,\ y = wu,\ v \to w$)
- regular grammars ($N \to aM \mid a \mid \varepsilon$),
- two-way automata,
- read-only Turing machines,
- weak second-order monadic logic of one successor (wS1S),
- *et un raton laveur*. . .

. . . though it should be noted that "equivalent in power" does not translate to "with the same algorithmic complexity".

We are looking for a specification language for properties; we want to translate things like "Z or if X then Y". We are looking for a formal *logic*. Thus our first candidate as a specification language is wS1S.

11.1.1 What Does that Word Salad Even Mean?

(w)S1S is the (weak) monadic second order theory of $\mathbb{N}$ with one successor.

That is quite a lot of big impressive words. Let's break it down:

(1) **second order theory:** a logic with $\forall$, $\exists$, and all the Boolean operators, that can quantify over elements ($\forall x,\ P(x)$) or sets ($\forall X,\ X \subseteq Y$), or even functions and relations, e.g.

$$\exists R,\ \forall x, y, z,\ (xRy \wedge yRz \implies xRz)\ .$$

First order would quantify on elements only.

(2) **monadic:** only quantifies on elements and sets (which are *monadic* predicates $P(x)$, cf. the set-predicate bijection already mentioned in this course); may not quantify over relations.

(3) **of $\mathbb{N}$:** our elements that we quantify over are members of $\mathbb{N}$. This is arguably a consequence of the next point.

We are going to see those elements as *positions* in a word. Recall that a word $w \in \Sigma^*$ of length n is a function $w : [\![1, n]\!] \to \Sigma$. (Though we are going to use $w : [\![0, n-1]\!] \to \Sigma$ in this class.)

(4) **with one successor:** intuitively, $\mathbb{N}$ is not *just* a set of elements. It has a strong underlying algebraic *structure* — multiple structures in fact. When you study a logic over a set, you must specify what structure it has direct access to for this set. In that case, we only give access to the basic, inductive structure of $\mathbb{N}$. Recall that $\mathbb{N}$ is defined inductively by the axioms

$$\frac{}{0 \in \mathbb{N}} \quad \text{and} \quad \frac{n \in \mathbb{N}}{n+1 \in \mathbb{N}}\,,$$

where $n+1$ is not to be understood as the application of some binary operator $+$, but only as a syntax for "the successor of n". Addition is then defined inductively *from* those axioms. Our usual notations are shortcuts:

$$\begin{aligned} 1 &= 0+1 \\ 2 &= (0+1)+1 \\ 3 &= ((0+1)+1)+1 \\ \dots \end{aligned}$$

We have no other given access to any structure of $\mathbb{N}$. If we want to say "$x \geqslant y$", we have to redefine that inside the logic, using only the basic relation "x is successor to y" and the other tools provided by the logic. Depending on these other tools, *the logic may or may not be expressive enough for that.*

There are other logics (w)S2S, …,(w)SkS, which correspond to tree-like structures instead of words. For instance, S2S is the logic of *two* successors, which we can interpret as a logic on positions in binary trees:

$$\frac{}{\varepsilon \in \mathbb{T}} \quad \text{and} \quad \frac{p \in \mathbb{T}}{p0 \in \mathbb{T} \quad p1 \in \mathbb{T}}\,,$$

where ε is the root and $p0$, $p1$ are the left and right children of p. Those logics have properties and complexities very similar to those of S1S.

(5) **with letter predicates:** this is not mentioned explicitly, but the only other structure information provided to the logic besides $+1$ are unary predicates $a, b, \dots$ for each letter of a given alphabet Σ. $a(x)$, or $x \in a$ if we view them as sets, means "at position x in the word, the letter is a". Thus $a(0)$ means "the first letter is a".

(6) **weak**: quantifications is over finite sets only. That is, $\forall X$ means "for every finite set X". This means that the weak variant of S1S deals with finite words, which have finite sets of positions, whereas the strong version deals with infinite words.

wS1S corresponds to the regular languages, and S1S to the ω-regular languages, i.e. regular languages of infinite words.

ω-regular languages are defined from the same two fundamental operators as regular languages: $L \cup M$ and LM (concatenation), with only the third changing. Whereas regular languages have L^*, the Kleene star, finite repeated concatenation of L onto itself, ω-regular languages have L^ω, infinite repeated concatenation of L onto itself.

They are equivalent to automata called Büchi automata, which are basically NFA with a different accepting condition: an infinite word is accepted if and only if there is a run in which one of the infinitely occurring states is final.

Despite their similarities to NFA, not *all* the nice properties of NFA can be lifted to Büchi automata. For instance deterministic Büchi automata are strictly weaker than their non-deterministic counterparts.

In order to avoid getting sidetracked and save some time, we shall not go farther on ω-languages in this course, and keep the notion of infinite word intuitive. You *will* definitely encounter such things in the literature, though. In fact most books and lectures on those topics jumpt straight to Büchi automata, given that reactive systems are the main target for model-checking.

11.1.2 Formal Syntax and Semantics

With this out of the way, let us write down the syntax and semantics of this logic. Let Σ be our underlying (finite) alphabet. Let $\mathbb{x} = x, x_1, x_2, \dots, y, \dots$ be our first-order variables, and $\mathbb{X} = X, X_1, X_2, \dots$ be our second-order variables, which here just means they are set variables. Of course we assume $\mathbb{x} \cap \mathbb{X} = \emptyset$. We have the syntax:

$$\varphi \in \text{(w)S1S} \quad ::= \quad a(x) \mid x = y + 1 \mid x \in X \mid \exists x : \varphi \mid \exists X : \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \,.$$

Of course, this syntax can be extended with $\forall$, $\vee$, etc, but since they can be expressed in terms of the other elements with the usual semantics, e.g. $p \vee q = \neg(\neg p \wedge \neg q)$, we don't bother mentioning them to save time and space and avoid, in the end, repeating ourselves.

Speaking of semantics, the semantics of (w)S1S is defined as follows, for any word $w \in \Sigma^*$ (or $w \in \Sigma^\omega$), and **interpretations**, or *valuations* of the variables

$$I_1 \in \mathbb{x} \nrightarrow \mathbb{N}, \quad I_2 \in \mathbb{X} \nrightarrow \wp(\mathbb{N}), \quad I = I_1 \cup I_2 \, .$$

We have:

$$\begin{array}{lll}
w, I \models a(x) & \Leftrightarrow & w\big(I(x)\big) = a \\
w, I \models x = y + 1 & \Leftrightarrow & I(x) = I(y) + 1 \\
w, I \models x \in X & \Leftrightarrow & I(x) \in I(X) \\
w, I \models \exists x : \varphi & \Leftrightarrow & x \notin \operatorname{dom} I,\ x \text{ free in } \varphi,\ \exists p \in \operatorname{dom} w : \\
 & & \quad w, I \cup \{x \mapsto p\} \models \varphi \\
w, I \models \exists X : \varphi & \Leftrightarrow & X \notin \operatorname{dom} I,\ X \text{ free in } \varphi,\ \exists P \subseteq \operatorname{dom} w : \\
 & & \quad w, I \cup \{X \mapsto P\} \models \varphi \\
w, I \models \varphi \wedge \psi & \Leftrightarrow & w, I \models \varphi \ \wedge\ w, I \models \psi \\
w, I \models \neg\varphi & \Leftrightarrow & w, I \not\models \varphi
\end{array}$$

Furthermore, for $\varphi \in$ (w)S1S we let

$$[\![\varphi]\!] \;=\; \big\{\, w \in \Sigma^* \;\big|\; w, \varnothing \models \varphi \,\big\}$$

be the **language described by φ**.

END FIFTH LECTURE (2021–2022)

11.1.2.1 *An Example*

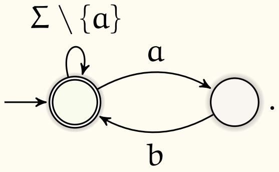
Let us consider a very simple property, of the form "every request a is immediately followed by a response b". This is coded by the following formula:

$$\varphi \;\equiv\; \forall x,\ a(x) \implies \big(\exists y : y = x + 1 \wedge b(y)\big) \,,$$

which we are going to write more concisely as

$$\varphi \;\equiv\; \forall x,\ a(x) \Rightarrow b(x+1) \,.$$

The set of words that satisfy this property is also described / accepted by the following automaton:

$A =$  .

In other words, φ and A are equivalent, which is to say that we have

$$[\![\varphi]\!] \;=\; [\![A]\!] \,.$$

As mentioned before, this is not an isolated case. For every wS1S formula there is an equivalent automaton, and for every automaton there is an equivalent formula. Those are the Büchi and Thatcher–Wright Theorems, which we shall partially prove later on. We will only show that for every automaton there is an equivalent formula, and admit the rest.

Before that, we will need to extend the syntax of our logic a bit, as it is quite unwieldy in its minimalistic form. Minimalism is useful for proofs (the fewer cases we have, the better), but syntactic sugar makes the medicine go down in day-to-day use.

11.1.3 Extending the Syntax; Writing Properties

Extending the syntax of wS1S also serves as a series of exercises of the form "how do I express *that* notion with this logic?". You would be well advised to treat the following points as such, and practice using the logic by proposing your own solutions before reading mine.

For each notion we introduce the usual notation (or some obvious notation for the more exotic concepts), and give a corresponding statement in wS1S. Those are to be seen as "macros"; you can use the notations, replacing them with the corresponding statement to obtain a wS1S formula following the minimalistic syntax.

Of course, we may reuse existing macros to define new ones. . .

(1) Boolean operators:

$$p \vee q \;\equiv\; \neg(\neg p \wedge \neg q) \qquad p \Rightarrow q \;\equiv\; \neg p \vee q \qquad p \Leftrightarrow q \;\equiv\; (p \Rightarrow q) \wedge (q \Rightarrow p)$$

(2) Universal quantifier:

$$\forall x,\; P(x) \;\equiv\; \neg \exists x : \neg P(x)$$

(3) Quantification in a set:

$$\forall x \in X,\; P(x) \;\equiv\; \forall x,\; x \in X \Rightarrow P(x)$$

$$\exists x \in X,\; P(x) \;\equiv\; \exists x,\; x \in X \wedge P(x)$$

(4) Root/Zero position:

$$x = 0 \;\equiv\; \neg \exists y : x = y + 1$$

(5) Property of successor position:

$$P(x+1) \;\equiv\; \exists y : y = x + 1 \wedge P(y)$$

(6) Property of predecessor position:

$$P(x-1) \;\equiv\; \exists y : x = y + 1 \wedge P(y)$$

Note that $P(x-1)$ will always be false if $x-1$ is undefined — that is, if $x = 0$ — regardless of P.

(7) Element equality: (often given as part of the structure; here we redefine it)

$$x = y \quad\equiv\quad \forall X,\ x \in X \Leftrightarrow y \in X$$

(8) Set emptiness:

$$X = \emptyset \quad\equiv\quad \forall x,\ \neg(x \in X)$$

We will also write $x \notin X$ for $\neg(x \in X)$.

(9) Set inclusion:

$$X \subseteq Y \quad\equiv\quad \forall x,\ (x \in X \Rightarrow x \in Y)$$

(10) Set equality:

$$X = Y \quad\equiv\quad X \subseteq Y \ \wedge\ Y \subseteq X$$

(11) Set intersection and union (binary):

$$Z = X \cap Y \quad\equiv\quad \forall x,\ x \in Z \Leftrightarrow (x \in X \wedge x \in Y)$$

$$Z = X \cup Y \quad\equiv\quad \forall x,\ x \in Z \Leftrightarrow (x \in X \vee x \in Y)$$

(12) Set intersection and union (n-ary):

$$Z = \bigcap_{k=1}^{n} X_k \quad\equiv\quad \forall x,\ x \in Z \Leftrightarrow \bigwedge_{k=1}^{n} x \in X_k$$

$$Z = \bigcup_{k=1}^{n} X_k \quad\equiv\quad \forall x,\ x \in Z \Leftrightarrow \bigvee_{k=1}^{n} x \in X_k$$

(13) Partition of a set:

$$X_1, \ldots, X_n \text{ partition } Z \quad\equiv\quad Z = \bigcup_{k=1}^{n} X_k \ \wedge \bigwedge_{\substack{i,j \in [\![1,n]\!] \\ i<j}} X_i \cap X_j = \emptyset$$

Digression / Question: Here we used previously defined syntax “macros” for union and intersection. The replacement of the union is straightforward, but we have patterns of the form $X_i \cap X_j$ and not exactly the previously defined $Z = X \cap Y$. Do you see how to rewrite the formula to accommodate that?

Answer: use $\exists$ to name the quantity you are interested in. A sub-formula of the form

$$P(X \cap Y)$$

can be rewritten as

$$\exists Z : Z = X \cap Y \ \wedge \ P(Z) \ .$$

The same patterns will apply to many other constructions. We already saw that with $P(x \pm 1)$. Of course you must be careful to use fresh variables when rewriting formulæ. In our current case, when partially rewritten, we obtain

$$\forall x,\ x \in Z \Leftrightarrow \bigvee_{k=1}^{n} x \in X_k \ \wedge \bigwedge_{\substack{i,j \in \llbracket 1,n \rrbracket \\ i \neq j}} \exists Z_{ij} : Z_{ij} = X_i \cap X_j \ \wedge \ Z_{ij} = \varnothing \ .$$

There remains to rewrite $Z_{ij} = X_i \cap X_j \ \wedge \ Z_{ij} = \varnothing$, and we obtain the *nearly* fully rewritten formula:

$$\forall x,\ x \in Z \Leftrightarrow \bigvee_{k=1}^{n} x \in X_k \wedge \bigwedge_{\substack{i,j \in \llbracket 1,n \rrbracket \\ i \neq j}} \exists Z_{ij} : \forall z,\ z \in Z_{ij} \Leftrightarrow (z \in X_i \wedge z \in Y_j) \wedge \forall z,\ z \notin Z_{ij} \ .$$

Technically, at that point, we would also need to rewrite $\vee$ and $\Leftrightarrow$ if we are sticking to the minimalist syntax, but I hope you get the idea by now.

The point is not to rewrite formulæ for the pleasure, but to become convinced that we can take calculated liberties with its syntax for our convenience without fundamentally altering its expressive power.

Some parts of the proofs of the Büchi and Thatcher–Wright Theorems (parts which we are not going to see) actually use wS1S_0, an even more restrictive syntax than what we used to define the logic, because it makes the proofs simpler.

The message I hope to convey here is that a given logic can come in many syntaxes, and that you should not be excessively surprised if you find heavy variations in the literature; what matters is the basic structure, the operators, and whether they can be rewritten in terms of one another.

(14) Set X is a singleton:

$$\operatorname{sing}(X) \quad \equiv \quad X \neq \varnothing \ \wedge \ \forall Y,\ Y \subseteq X \Rightarrow (Y = X \ \vee \ Y = \varnothing)$$

(15) Set X is closed above:

$$\text{ClosedAbove}(X) \quad \equiv \quad \forall x,\ x \in X \Rightarrow x + 1 \in X$$

(16) Inequalities:

$$x \leqslant y \quad \equiv \quad \forall X : \big(x \in X \ \wedge \ \text{ClosedAbove}(X)\big) \Rightarrow y \in X$$

$$x < y \quad \equiv \quad x \leqslant y \wedge x \neq y \qquad x \geqslant y \quad \equiv \quad \neg(x < y) \qquad x > y \quad \equiv \quad \neg(x \leqslant y)$$

(17) Ranges:

$$x \in \llbracket i, j \rrbracket \quad \equiv \quad i \leqslant x \leqslant j \quad \equiv \quad i \leqslant x \ \wedge \ x \leqslant j$$

11.1.4 The Büchi and Thatcher–Wright Theorems

Those important theorems show the equivalence of the (w)SkS logics and corresponding classes of automata. Historically, this is how those logics were shown to be decidable.

For us, it means that our properties can be "compiled" into automata, and thus interact with the models of our systems for the purpose of verifying their compliance.

Theorem 1 ([Büchi, 1960]). *A language is recognizable by a Büchi automaton (resp. a NFA) if and only if it is definable in S1S (resp. wS1S).*

Theorem 2 ([Thatcher and Wright, 1968]). *Büchi's theorem generalises to (w)SkS and tree automata.*

11.1.4.1 *For every NFA, there is an equivalent wS1S formula*

Let us show half of Büchi's theorem, in the case of NFA: for every NFA $\mathcal{A} = \langle \Sigma, Q, I, F, \Delta \rangle$, there is an equivalent formula φ.

We shall need a predicate to identify the end position of the word. We denote by $\perp$ the "frontier" position just after the word is ended; for instance, for the word abc, we have $\perp = 3$:

$$\underbrace{a}_{0}\underbrace{b}_{1}\underbrace{c}_{2}\underbrace{\quad}_{\perp} .$$

We can obtain $\perp$ via, first, a predicate to detect "out-of-bounds" positions:

$$\mathrm{out}(x) \quad \equiv \quad \neg \bigvee_{a \in \Sigma} a(x) ,$$

and then, finding the first out-of-bounds position:

$$x = \perp \quad \equiv \quad \mathrm{out}(x) \ \wedge \ (x = 0 \ \vee \ \neg\mathrm{out}(x-1)) .$$

We build φ following the idea that we encode a successful run of $\mathcal{A}$ on a word; to each state q of the automaton will correspond a variable of φ, which will contain the set of positions where the automaton is in state q in the run.

For instance, a run

$$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_1 \xrightarrow{c} q_0$$

is encoded by the set variables

$$q_0 = \{0, 3\} \quad \text{and} \quad q_1 = \{1, 2\} ,$$

without forgetting that the structure itself also provides us with the predicates / set variables encoding the words; for instance for abc we have:

$$a = \{0\} \quad \text{and} \quad b = \{1\} \quad \text{and} \quad c = \{2\} .$$

We have finally:

$$\varphi \;\equiv\; \mathop{\Large\exists}_{q\in Q} q : \left(0 \in \bigcup I\right) \wedge \left(\bot \in \bigcup F\right) \wedge \forall x \in [\![1, \bot]\!], \bigwedge_{q\in Q} \left(x \in q \Leftrightarrow \bigvee_{p \xrightarrow{a} q \in \Delta} x - 1 \in p \cap a \right) .$$

Just in case it's not clear to everyone, $\bigcup X$ is a common notational shortcut for $\bigcup_{x\in X} x$.

Note that this formula is, *in fine*, little more than a formalisation of the notion of "successful run" for an NFA.

> **A few notes on this.** The proof of Büchi for finite words is not often found in the literature. I either find proofs for wSkS (finite trees), or proofs for S1S (ω-words), but not for wS1S. It does not help that the seminal papers are quite old, and not easily accessible.
>
> Nevertheless, what I have presented here is a direct adaptation of Büchi's proof for ω-languages. If someone finds the seminal paper for wS1S ↔ NFA equivalence, please send a reference my way; or better yet a PDF.

The proof of the same half of the Thatcher–Wright Theorem is extremely similar to that, but for nondeterministic tree automata instead of NFA.

11.1.4.2 *For every wS1S formula, there is an equivalent NFA*

The second part of the proof, going from formulæ to automata, is longer and more technical, and will not be covered in these lectures. We **admit** that result.

We shall, however, need to talk about the algorithmic complexity of this transformation.

11.1.5 Algorithmic Complexity and Suitability for Verification

An important question for any logic is its decidability, a term which in this context is shorthand for decidabilty of its satisfiability problem. That is to say, given a formula φ, is there some valuation that actually satisfies it?

In other words, that is the question

$$[\![\varphi]\!] \;=\; \varnothing?$$

For most logics, this is a hard question; you should know at this point that, for the basic propositional Boolean logic, this question is the archetypal NP-complete problem: SAT. If the question is hard for propositional logic, is it even decidable for much more powerful logics like wS1S? The answer is far from obvious, especially given that first-order logic, in all generality, is undecidable — *a fortiori* higher-order logics are as well. But of course, we consider here a very specific and restricted second-order logic.

This is why the Büchi and Thatcher-Wright theorems are so important: by providing a transformation from logic to NFA, they prove that the logics are decidable, for indeed

$$[\![\varphi]\!] = \emptyset \quad \Leftrightarrow \quad [\![A_\varphi]\!] = \emptyset \,.$$

Furthermore, since emptiness testing for NFA is linear-time — it is just an accessibility problem: are final states reachable from initial states? — the complexity of the satisfiability test is determined by the size of A_φ, as a function of the size of φ. A lower-bound on the complexity of wS1S's satisfiability implies a lower bound on the size of A_φ in the worst case.

This is important to us, since we need to compute whether the system, for which we have an automaton, stays within the bounds of the specification, given by a formula φ:

$$[\![A_S]\!] \subseteq [\![A_\varphi]\!] \,.$$

Concretely this computation will be effected as

$$[\![A_S]\!] \cap [\![\overline{A_\varphi}]\!] = \emptyset \,.$$

which means we need an automaton to complement and involve in a product. The size of A_φ needs to be reasonable for this programme to be applicable in practice.

Unfortunately, it turns out that satisfiability of (w)S1S is non-ELEMENTARY, and this is a lower and upper bound. To fully appreciate what that mean, a brief reminder of the important complexity classes follows.

11.1.5.1 Reminders about Complexity Theory

Figure 1[p90] [i] offers a quick reminder of the hierarchy of computational complexity classes.

$\mathsf{DTIME}(f(n))$ is the class of problems solvable by a deterministic Turing machine in time $O(f(n))$, where n is the size of the input. Likewise, $\mathsf{NTIME}(f(n))$ is the set of decision problems that can be solved by a non-deterministic Turing machine in $O(f(n))$ time. Similarly $\mathsf{DSPACE}(f(n))$ and $\mathsf{NSPACE}(f(n))$ are the sets of decision problems that can be solved by a deterministic (resp. non-deterministic) Turing machine in $O(f(n))$ space.

Determinism is less important for space than for time, by Savitch's theorem: for any function $f \in \Omega(\log(n))$, i.e. for any function f bounded below by log asymptotically,

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DSPACE}\left(f(n)^2\right) \,. \qquad \text{(Savitch 1970)}$$

The main classes are as follows:

$$\mathsf{P} \quad = \quad \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}(n^k) \quad = \quad \mathsf{DTIME}(n^{O(1)})$$

[i] Slight modifications on a figure by Sebastian Sardina.

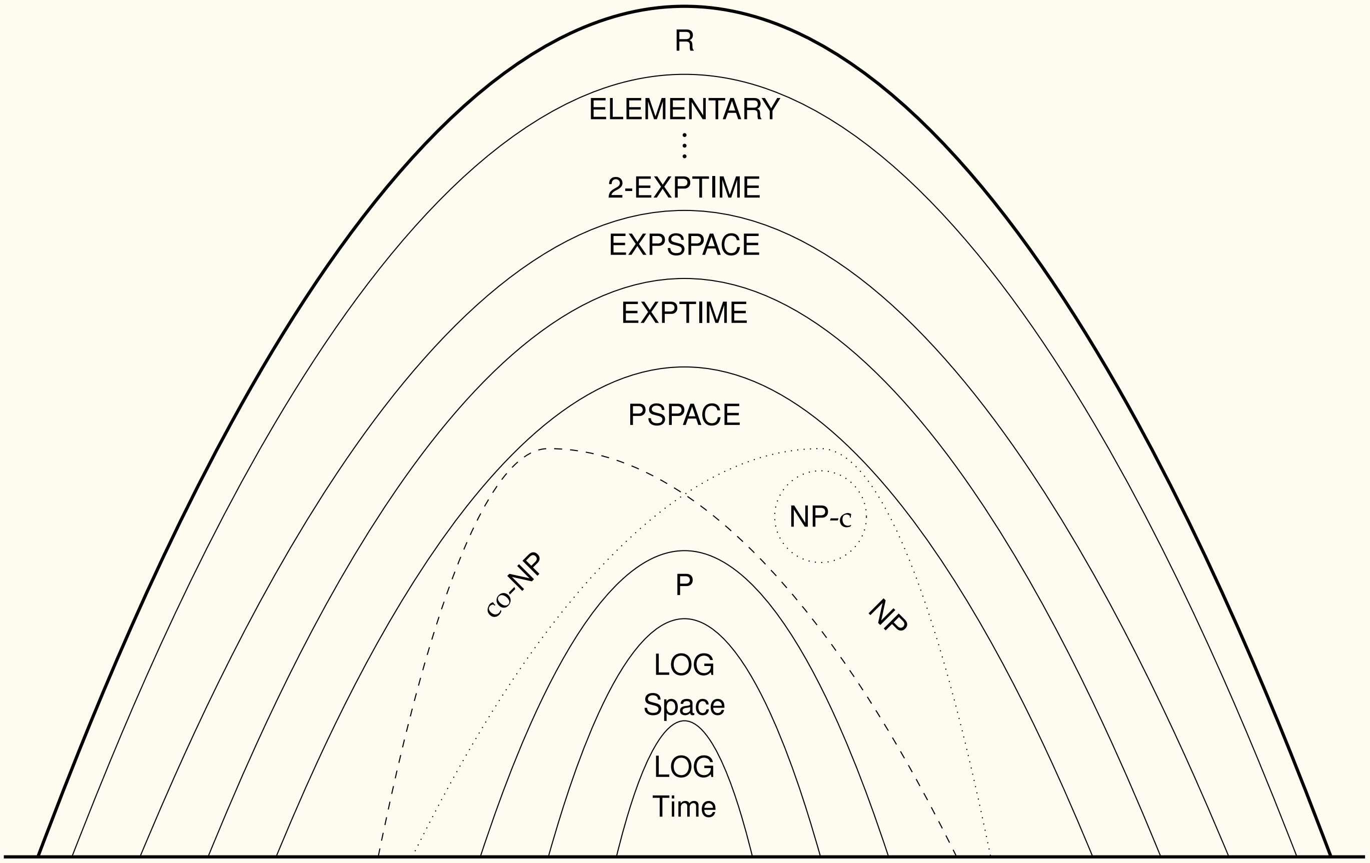


Figure 1: Computational Complexity Classes.

$$\begin{aligned}
\mathsf{NP} &= \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(n^k) = \mathsf{NTIME}(n^{O(1)}) \\
\mathsf{PSPACE} &= \bigcup_{k\in\mathbb{N}} \mathsf{DSPACE}(n^k) = \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}(n^k)
\end{aligned}$$

Note that determinism does not matter for this class, or any space-related class above, because (Savitch 1970) means that non-determinism is simulated by deterministic machines with a quadratic space cost.

$$\begin{aligned}
\mathsf{EXPTIME} &= \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}\left(2^{n^k}\right) \\
\mathsf{NEXPTIME} &= \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(2^{n^k}) \\
\mathsf{EXPSPACE} &= \bigcup_{k\in\mathbb{N}} \mathsf{DSPACE}\left(2^{n^k}\right) = \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}\left(2^{n^k}\right) \\
\mathsf{2\text{-}EXPTIME} &= \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}\left(2^{2^{n^k}}\right)
\end{aligned}$$

This generalises to higher and higher towers of exponentials:

$$\mathsf{N\text{-}EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}\left(\underbrace{2^{2^{2^{\cdots^{2^{n^k}}}}}}_{\text{Tower of size } N}\right),$$

And then we have

$$\begin{aligned}
\mathsf{ELEMENTARY} &= \bigcup_{k\in\mathbb{N}} k\text{-}\mathsf{EXPTIME} \\
&= \mathsf{DTIME}(2^n) \cup \mathsf{DTIME}(2^{2^n}) \cup \mathsf{DTIME}\left(2^{2^{2^n}}\right) \cup \cdots,
\end{aligned}$$

the class of problems decidable in time that is bounded by tower of exponentials of arbitrary height. This is our stop.

11.1.5.2 *What Does it Mean for Our Purposes?*

The automaton A_φ turns out to be of size *not* bounded by *any* tower of exponential of constant height. That is to say, you can always find a formula that requires a higher tower. The height of the tower is a function of how many times certain patterns — quantifier alternation, in this case — appear in the formula.

Unfortunately I haven't covered this part of the proof in the lectures, but the rough idea is that, whenever you have nested patterns of the form $\forall x,\ \exists y : \cdots$, you *have* to determinise the automaton for the subformula. Each time you do that, it may blow up exponentially. Since there is no limit upon the depth of such patterns in the input formula, there is no bound upon the height of the exponential tower.

This is not a matter of finding better algorithms either. There exist formulæ for which it is proven that no smaller automaton is suitable.

Of course, simpler formulæ do not suffer from those problems, but, combined with the arguable lack of intuitiveness of the logic, this makes (w)S1S unattractive as a specification language for the purpose of verification. And thus research moved on, to temporal logics — which is the object of the next section.

11.2 CTL*: Computation Tree and Linear Time Logic (CTL+LTL+$\cdots$)

The temporal logics CTL (Computation Tree Logic) [Clarke and Emerson, 1981], LTL (Linear Time Logic) [Pnueli, 1977], and CTL* (a superset of both) [Emerson and Halpern, 1983], are modal logics designed specifically with two goals in mind

(1) Efficient verification algorithms

(2) More user-friendly syntax and semantics for the purpose of specifying the kind of properties we are interested in. At the end of the day they are still modal logics, so don't expect immediate clarity…

To summarise their relationships very quickly, LTL was proposed first, and is probably the most used nowadays. That was not always the case, though. CTL, which was proposed to express some properties impossible to express in LTL, turned out to have a very efficient verification algorithm, and gained considerable traction. LTL's complexity is worse, but modern systems mitigate this to a large extent, and it has some distinct advantages over CTL (fairness properties, easier semantics,…) CTL and LTL are incomparable, in that each expresses properties that are beyond the other's power.

CTL* is a strict superset of both CTL and LTL. Despite its asymptotic complexity for the model-checking problem being the same as LTL's, it *is* more complex — in the vernacular sense — than either CTL or LTL, and is not, to my knowledge, widely used in practical and industrial applications. It is, however, a good tool for theoretical study.

All of those are strict subsets of S1S with respect to expressive power. LTL is equivalent to FO[$<$], the monadic first order logic of order. CTL* is strictly less powerful than $\text{FO}^2(*)$, first order logic of two variables with transitive closure. Those are *small* fragments of S1S. For our purposes, they are quite enough, and constitute a good compromise between expressive power, simplicity, and algorithmic efficiency.

Most lectures and books on the subject keep CTL* for last, if they mention it at all. I choose to begin by it because CTL and LTL will then be defined as restrictions to that, and I do not find CTL in any way easier to understand than CTL*. Be mindful, if you read other sources, not to confuse CTL and CTL*.

11.2.1 Kripke Structures and Paths

CTL* evaluates whether the sequences of states taken by an automaton during its execution satisfy some properties. Let us get some definitions out of the way:

Let A be an automaton with the usual notations. We see it a transition system, really; we don't care much about final states or transition labels in this case. Furthermore, we embrace the "reactive system" aspect, and assume there is a transition starting from each states. What matters is whether you can go from state p to state q, and you never have to stop. This is often called a **Kripke structure** rather than an automaton.

We are interested in properties of states, so let us define once and for all a set $\mathbb{A}$ of **atomic properties** of the states. For instance, they can be of the form "the intruder has access", or "a request for access has been made", or "a request for access has been granted", or "$x = 0$". They are the basic building blocks of our statements, which we shall put in relation with each other with respect to time. For instance, "never does the intruder have access", and "whenever a request has been made, eventually it is granted".

Those atomic properties are associated to states of the system via a **labelling function**

$$\ell : Q \to \wp(\mathbb{A}) \,,$$

associating to each state of the system the set of atomic properties which that states satisfies.

A **path** (or *run*, or *execution*, or *trace*...) is a word $\pi \in Q^\omega$, such that for all $k \in \mathbb{N}$, $\pi[k] \to \pi[k+1] \in \Delta$. We let

$$\Pi(q) \quad = \quad \left\{ \pi \in Q^\omega \;\middle|\; \pi[0] = q \;\wedge\; \forall k \in \mathbb{N},\ \pi[k] \to \pi[k+1] \in \Delta \right\}$$

be the set of **paths starting in state** q. We shall use Python-like index and slice notation, 0 being the first index as usual.

END SEVENTH LECTURE (2020–2021)

11.2.2 Syntax and Semantics of CTL*

There are two kinds of formulæ in CTL*: **state formulæ**, which are the "entry point", so to speak, and are therefore simply called "CTL* formulæ", and **path formulæ**.

The (minimalist) syntax is as follows:

$$\begin{array}{lcll} \varphi \in \text{CTL* (state)} & ::= & p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists\psi & \qquad p \in \mathbb{A} \\[1em] \psi \in \text{CTL* (path)} & ::= & \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \circ\psi \mid \psi \,\mathbf{U}\, \psi & \end{array}$$

Here, $\exists$ is pronounced "for some path" or "there is a path such that" (it is often written **E** in the literature); the temporal modality $\circ$ is pronounced "next" (and is often written **X** in the literature). The temporal modality **U** is pronounced "until". Note that it is the letter U, and not a union symbol $\bigcup$.

Of course, there are other operators and modalities that can be defined from those building blocks, as usual. Let us first give the semantics of the above.

A state $q \in Q$ satisfies a state formula φ under the following conditions:

$$\begin{array}{lllcl}
q & \models & p & \Leftrightarrow & p \in \ell(q) \\
q & \models & \neg\varphi & \Leftrightarrow & q \not\models \varphi \\
q & \models & \varphi \land \varphi' & \Leftrightarrow & q \models \varphi \ \land \ q \models \varphi' \\
q & \models & \exists\psi & \Leftrightarrow & \exists\pi \in \Pi(q) : \ \pi \models \psi
\end{array}$$

A path $\pi \in Q^\omega$ satisfies a path formula ψ under the following conditions:

$$\begin{array}{lllcl}
\pi & \models & \varphi & \Leftrightarrow & \pi[0] \models \varphi \\
\pi & \models & \neg\psi & \Leftrightarrow & \pi \not\models \psi \\
\pi & \models & \psi \land \psi' & \Leftrightarrow & \pi \models \psi \ \land \ \pi \models \psi' \\
\pi & \models & \circ\psi & \Leftrightarrow & \pi[1:] \models \psi \\
\pi & \models & \psi \,\mathbf{U}\, \psi' & \Leftrightarrow & \exists k \in \mathbb{N} : (\forall i \in [\![0, k[\![, \ \pi[i:] \models \psi) \ \land \ \pi[k:] \models \psi'
\end{array}$$

As mentioned above, we have for now a minimal set of operators and modalities, which we extend by adding the usual Boolean values and operators $\top, \bot, \lor, \Rightarrow, \iff$, etc, for both path and state formulæ, as well as, for path formulæ:

$$\Diamond\psi \ \equiv \ \top \,\mathbf{U}\, \psi \quad \text{and} \quad \Box\psi \ \equiv \ \neg\Diamond\neg\psi \,.$$

The temporal modality $\Diamond$ is read "eventually / finally[j]" (and often written **F** in the literature) while $\Box$ is read "globally", or "always", (and often written **G** in the literature). For state formulæ, we add universal quantification over paths:

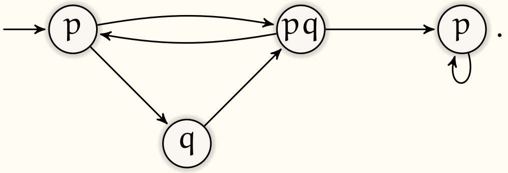
$$\forall\psi \ \equiv \ \neg\exists\neg\psi \,.$$

This path quantifier is often written **A** in the literature.

11.2.3 A Few Examples, to Help the Semantics go Down

Let us take a simple example Kripke structure (cribbed from [Baier and Katoen, 2008]), and run a few formulæ on it.

We take $\mathbb{A} = \{p, q\}$ and the labelled Kripke structure



If you refer to `test.py`, you will find this structure implemented as

[j] *fatalement* in French

```
kat = NFA.spec("""
0
--
0 1 2
1 0 3
2 1
3 3
""", name="Katoen examples",style="ts").visu()

labels = { 0:{p}, 1:{p,q}, 2:{q}, 3:{p} }
```

Now let us examine a few formulæ, and for each one, colour in green the nodes that satisfy it. For CTL formulæ — we shall see the detail of the LTL and CTL restrictions of CTL* later, once we have intuitions on the semantics; for now the general idea is that quantifiers and temporal modalities are always bundled together in CTL, whereas LTL formulæ are of the form $\forall\psi$, with no quantifier in ψ — which make up most of the examples, I have implemented the model-checking algorithm in `ctl.py`.

For instance,

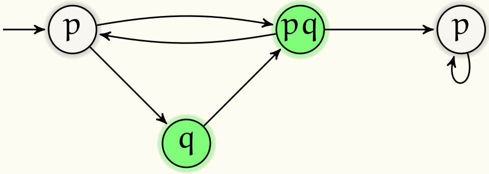
```
checkvisu( kat,labels,
  (EU, p, (AND, (NOT, p), (AU, (NOT, p), q))),
  visu=("simple","detailed") )
```

offers two different visualisations of the model-checking of our system `kat`, with labelling dictionary `labels`, for the formula

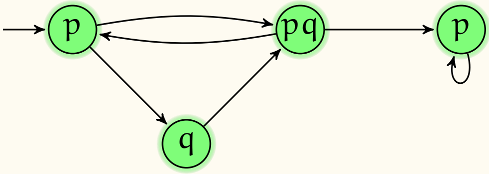
$$\exists\Big(p \,\mathbf{U}\, (\neg p \wedge \forall(\neg p \,\mathbf{U}\, q))\Big) .$$

Demo in `test.py` for complicated (CTL) examples.

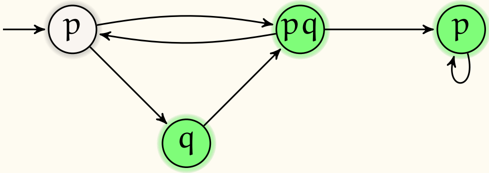
(1) q:



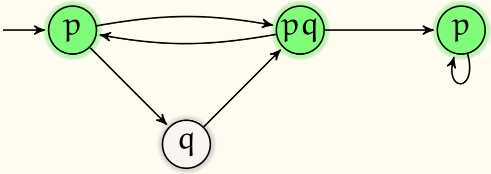
(2) $\exists\circ p$:



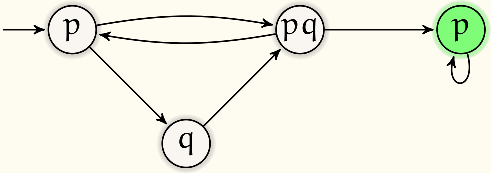
(3) $\forall\circ p$:



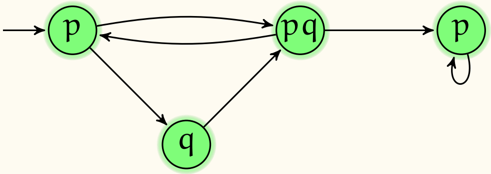
(4) $\exists\square p$:



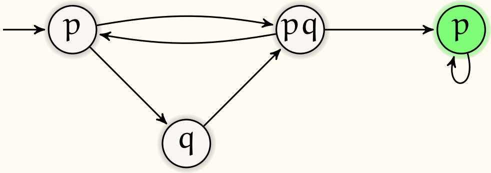
(5) $\forall\square p$:



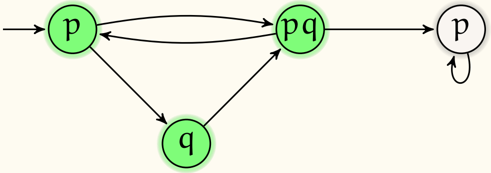
(6) $\exists\Diamond\exists\square p$
$\exists\Diamond\forall\square p$
$\forall\Diamond\exists\square p$



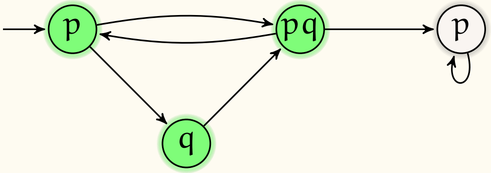
(7) $\forall\Diamond\forall\square p$:



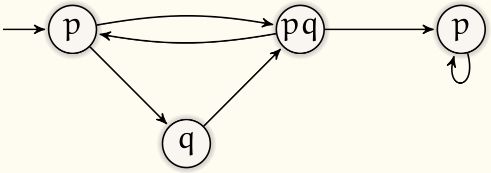
(8) $\forall(p\ \mathbf{U}\ q)$
$\exists(p\ \mathbf{U}\ q)$



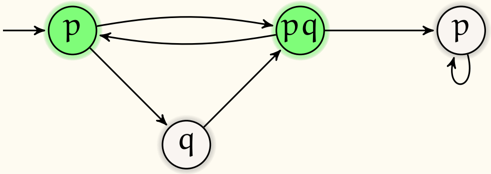
(9) $\exists\Big(p\ \mathbf{U}\ \big(\neg p \wedge \forall(\neg p\ \mathbf{U}\ q)\big)\Big)$:



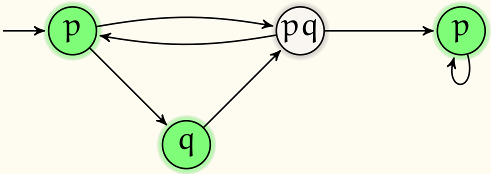
(10) $\forall\circ\forall\circ q$
$\forall\circ\exists\circ q$



(11) $\exists\circ\forall\circ q$
$\exists\circ\exists\circ q$



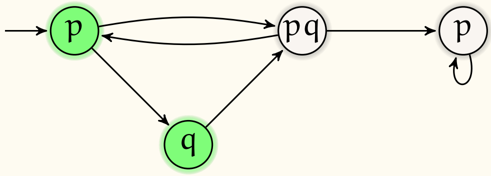
(12) $\forall\circ\circ p$ (that one is LTL):



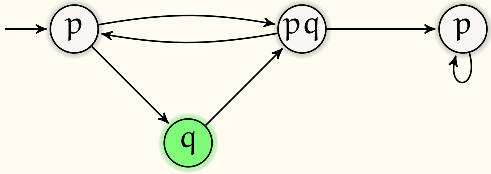
$\forall\circ\Diamond(p \land q)$ (LTL)

(13) $\forall\Diamond\circ(p \land q)$ (LTL)

$\forall\circ\forall\Diamond(p \land q)$



(14) $\forall\Diamond\forall\circ(p \land q)$:



11.2.4 LTL: Linear Time Logic

LTL formulæ express properties of the form "for all executions of the system, starting in the initial state, such and such holds". Thus, they correspond rather naturally to the following fragment of CTL*:

$$\varphi \;::=\; \forall\psi \qquad \psi \;::=\; p \mid \neg\psi \mid \psi \land \psi \mid \circ\psi \mid \psi \,\mathbf{U}\, \psi \qquad\qquad p \in \mathbb{A}$$

However, in LTL, the initial $\forall$ is not written. Thus LTL formulæ are simply CTL* path formulæ implicitly understood as universally quantified, and with no path quantifier.

Since we are here in the wider context of CTL*, in those lecture notes we shall still write LTL formulæ under the form $\forall\psi$, with the explicit quantifier, and understand that as the LTL fragment of CTL*.

Be aware that most of the literature on LTL will not do that. If you see a formula with a temporal modality that is *not* preceded by a path quantifier, such as $\Diamond p$, then it is a safe bet that it is an LTL formula, and should be interpreted as the CTL* formula $\forall\Diamond p$.

11.2.5 CTL: Computation Tree Logic

CTL is not quite as clean a restriction in terms of CTL*. Intuitively, it demands that each temporal modality be "bundled" with a path quantifier, and vice-versa.

We have the following syntax, as a restriction of CTL*:

$$\varphi \in \text{CTL (state)} \;::=\; p \mid \neg\varphi \mid \varphi \land \varphi \mid \exists\psi \qquad\qquad p \in \mathbb{A}$$

$$\psi \in \text{CTL (path)} \;::=\; \neg\psi \mid \circ\varphi \mid \varphi \,\mathbf{U}\, \varphi$$

Now let us see a more direct syntax:

$$\varphi \in \text{CTL} \;::=\; p \mid \neg\varphi \mid \varphi \land \varphi \mid \exists\circ\varphi \mid \exists[\varphi \,\mathbf{U}\, \varphi] \mid \forall[\varphi \,\mathbf{U}\, \varphi]$$

Those "bundles" are considered single operators; here, in this reduced syntax, we have:

- $\exists\circ$: "for some path, next"

- $\exists\mathbf{U}$: "for some path, until"; could be written as a standard binary operator $\varphi \exists\,\mathbf{U}\ \varphi$ instead of $\exists[\varphi\ \mathbf{U}\ \varphi]$ — while nobody would do that on paper, it is how those operators are coded in `ctl.py`, as it avoids implementing all the syntax of CTL*, and then having to syntactically validate well-formed CTL formulæ.
- $\forall\mathbf{U}$: "for all paths, until"

Other "bundled" operators can be added, beyond the usual Boolean extensions:

$$\begin{array}{rcll}
\exists\Diamond\varphi & \equiv & \exists[\top\ \mathbf{U}\ \varphi] & \text{“potentially holds”} \\
\forall\Diamond\varphi & \equiv & \forall[\top\ \mathbf{U}\ \varphi] & \text{“is inevitable”} \\
\exists\Box\varphi & \equiv & \neg\forall\Diamond\neg\varphi & \text{“potentially always”} \\
\forall\Box\varphi & \equiv & \neg\exists\Diamond\neg\varphi & \text{“invariantly”} \\
\forall\circ\varphi & \equiv & \neg\exists\circ\neg\varphi & \text{“for all paths next”}
\end{array}$$

Let us note that the "direct" version of the syntax we gave is a bit more restrictive than the first version, and works because of some equivalences. For instance the formula $\exists\neg\circ p$ is well-formed for the first syntax, but not for the second one. How, since it is equivalent to $\neg\forall\circ p$, this is not a problem. It as actually also equivalent to $\exists\circ\neg p$.

The direct version of the syntax is the more common in the literature. The merit of the first version is to showcase exactly in what respect it is subsumed by CTL*.

It is actually possible to extend CTL by allowing all standard Boolean operators ($\wedge, \vee, \neg$) in path formulæ, without changing its expressive power, thanks to equivalences such as

$$\begin{array}{rcl}
\exists\neg\circ p & \equiv & \exists\circ\neg p \\
\exists(\circ\varphi_1 \wedge \circ\varphi_2) & \equiv & \exists\circ(\varphi_1 \wedge \varphi_2) \\
& \ldots &
\end{array}$$

This is occasionally referred to as CTL+ [Baier and Katoen, 2008]. While the expressive power is the same, CTL+ formulæ can be much shorter than the equivalent CTL formulæ. Know that this is possible, but please write classical "direct syntax" CTL formulæ as much as possible.

11.2.6 Some Useful Properties and Miscellaneous Examples

11.2.6.1 *Mutual Exclusion*

Let C_1, C_2 be either the atomic properties "the process number 1 (resp. 2) is in its critical section", or state formulæ to the same effect (for instance, Boolean combinations of tests on relevant variables). Then mutual exclusion — the processes may not be in their critical sections at the same time — is given by

$$\forall\Box\big(\neg(C_1 \wedge C_2)\big)\ ,$$

which can be seen as both a CTL or LTL formula.

11.2.6.2 *Possible Access, Liveness, "infinitely often"*

The very weak liveness property "it is possible for process i to access its critical section" is written as

$$\exists\Diamond C_i \qquad \text{in CTL.}$$

This however does not mean that it has to. It may never access it in practice. This property is not expressible in LTL.

More strongly, we wan write

$$\forall\Diamond C_i \qquad \text{in LTL and CTL,}$$

ensuring that, no matter what, the process eventually accesses its critical section, at least once.

The strong liveness property "each process accesses its critical section infinitely often" can be expressed as

$$\forall\big(\Box\Diamond C_1 \ \wedge\ \Box\Diamond C_2\big) \qquad \text{in LTL}$$

or as

$$\forall\Box\forall\Diamond C_1 \ \wedge\ \forall\Box\forall\Diamond C_2 \qquad \text{in CTL.}$$

This is rather strong, and assumes that each process *wants* to enter its critical section infinitely often. A weakened form states that "every waiting process shall be granted access, infinitely often". Let W_i stand for "process i is waiting for critical access". We have:

$$\forall\Big(\big(\Box\Diamond W_1 \Rightarrow \Box\Diamond C_1\big) \ \wedge\ \big(\Box\Diamond W_2 \Rightarrow \Box\Diamond C_2\big)\Big) \qquad \text{in LTL}$$

11.2.6.3 *Requests Get Answers, Eventually*

The useful property "Every request (R) is eventually answered (A)" is written as

$$\forall\big(\Box(R \Rightarrow \Diamond A)\big) \qquad \text{in LTL}$$

$$\forall\Box\big(R \Rightarrow \forall\Diamond A\big) \qquad \text{in CTL.}$$

Note that "eventually" may take a looooong time. If the answer must come immediately, or, say, within k steps, we can write

$$\forall\Big(\Box\Big[R \Rightarrow \bigvee_{0\leqslant i\leqslant k} \circ^i A\Big]\Big) \qquad \text{in LTL,}$$

where $\circ^i$ means, of course, $\underbrace{\circ\circ\cdots\circ}_{i}$. Alternatively,

$$\forall\Box\Big(R \Rightarrow \bigvee_{0\leqslant i\leqslant k} (\forall\circ)^i A\Big) \qquad \text{in CTL.}$$

11.2.6.4 No Shoes, No Service

The property "Access (A) shall not be granted until the user wears proper shoes (S)" is easily expressed as

$$\forall(\neg A \,\mathbf{U}\, S)\,,$$

which is, conveniently, both a CTL and LTL formula.

Note that this formula enforces that the user *gets* the shoes, eventually, which may go beyond what was intended by the property. If you don't want that, a weaker variant is required:

$$\forall(\Box\neg A \;\vee\; \neg A \,\mathbf{U}\, S) \qquad \text{in LTL.}$$

This is directly covered by the **weak-until** modality, defined for LTL as

$$\varphi \,\mathbf{W}\, \psi \;\equiv\; \Box\varphi \;\vee\; \varphi \,\mathbf{U}\, \psi\,.$$

Note that this says nothing as to whether the user ever gets access after acquiring the proper shoes. Maybe he never requests access. Maybe the access-granting process is broken. Regardless, the property states a *necessary condition* for access, not a *sufficient* one.

Be careful not to translate more than what the property states. The property does not state "access *shall* be granted once the user gets shoes"...

11.2.6.5 Interdiction

The property "after an interdiction is issued (I), the event (E) does never happen again" is written

$$\forall\big(\Box(I \implies \Box\neg E)\big) \qquad \text{in LTL}$$

$$\forall\Box\big(I \implies \forall\Box\neg E\big) \qquad \text{in CTL.}$$

11.2.6.6 Necessary Steps

The French proverb "*c'est en forgeant qu'on devient forgeron*" (which has the general meaning "practise makes perfect"), is literally translated as "its is by constantly practising smithing (P) that one becomes a blacksmith (B)". This is a weaker type of property than "no shoes, no service", as no everybody eventually becomes a blacksmith. But *if they do*, they had to practise constantly before. Thus, the property is expressed as

$$\forall\big(\Diamond B \implies P \,\mathbf{U}\, B\big) \qquad \text{in LTL.} \tag{11.1}$$

That property is not expressible in CTL. Intuitively, in CTL it is hard to pin down a specific trace, and come back at it from different angles. For instance, the CTL formula

$$\forall\Diamond B \implies \forall[P \,\mathbf{U}\, B]$$

does not have the same meaning as (11.1). If there is *any* trace where you do *not* become a blacksmith, then the antecedent is false, and the property is true by default.

Another interesting proverb is "*Rien ne sert de courir, il faut partir à point*", meaning "a slow and steady start (S) wins the race (W)". Put another way, if you are going to win the race, you need to start slow. This applies to any number of races you might run in your life. This is expressed by

$$\forall\Box(\Diamond W \implies S) \qquad \text{in LTL.}$$

There again, that property is not expressible in CTL. Consider for instance

$$\forall\Box(\forall\Diamond W \implies S)\ .$$

A starting state from which there are paths that do not lead to victory would satisfy $\forall\Diamond W \implies S$ by default of the antecedent, and not be required to start in S. Any state with W would have all its paths contain W, and therefore would be required to have S at the same time.

11.2.7 Comparing LTL, CTL, and CTL*

As said earlier, CTL and LTL are incomparable, in that each expresses properties that are beyond the other's power, i.e. for which there exists no formula with equivalent semantics. They also have different algorithmic properties. In this section we shall go into a bit more detail about the differences between LTL, CTL, and CTL*.

11.2.7.1 Expressive Powers

11.2.7.2 Algorithmic Complexity

11.2.8 CTL Model-Checking Algorithm

11.3 Exercises on Logics and Automata

11.3.1 Exam 2020–2021

11.3.1.1 Problem Statement

In this exercise, we work in the context of finite words on $\Sigma = \{\, a, b, c, d\,\}$.

Furthermore, we interpret LTL path formulæ over finite words, viewing Σ as our set of atomic properties, and a finite word abc as the infinite path $\{a\}\{b\}\{c\}\varnothing^\omega$ (that is, we repeat $\varnothing$ indefinitely once the word is exhausted), and applying the usual semantics to that path. In that sense, an LTL path formula ψ defines the language $[\![\psi]\!] \subseteq \Sigma^*$ of the finite words that satisfy it.

Formally, we have a mapping m between finite words and paths:

$$m : \left|\begin{array}{ccc} \Sigma^* & \longrightarrow & \wp(\Sigma)^\omega \\ a_1, \dots, a_n & \longmapsto & \{a_1\}\dots\{a_n\}\varnothing^\omega \end{array}\right. ,$$

and we define

$$[\![\psi]\!] \quad = \quad \{ w \in \Sigma^* \mid m(w) \models \psi \} \,.$$

An automaton A, resp. a wS1S formula φ, is said to be equivalent to a path formula ψ if $[\![\psi]\!] = [\![A]\!]$, resp. $[\![\psi]\!] = [\![\varphi]\!]$.

(1) Give an automaton equivalent to the following LTL (path) formula:

$$\psi_1 \quad = \quad \Box(a \Rightarrow \circ\Box\neg a)$$

(2) Give a wS1S formula φ_1 equivalent to ψ_1.

(3) Let P_2 be the property "every occurence of d must immediately be followed by an "a", then a "b" ".

a. Give a wS1S formula φ_2 equivalent to this property.

b. Give an automaton A_2 equivalent to this property.

c. Give an LTL path formula ψ_2 equivalent to this property.

(4) Give an automaton equivalent to

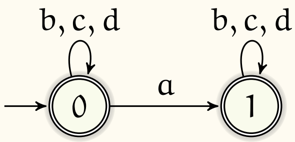
$$\psi_3 \quad = \quad \Box\big(a \Rightarrow \Diamond(b \land \circ b \land \Diamond c)\big) \,.$$

11.3.1.2 Solution

(1)

$$\Box(a \Rightarrow \circ\Box\neg a)$$

means "whenever there is an a, then from that point on there is never an a", or "there is at most one a", and thus corresponds to



Note: Many people completely ignored the second □, interpreting it as "the next symbol is not a", to the point that I wondered whether some may have experienced a PDF rendering bug. But no, many of those also copied the formula, correctly. This is one of those strange, and strangely common, mistakes, where I wonder whether the idea may have circulated through Discord or other channels. . .

(2)

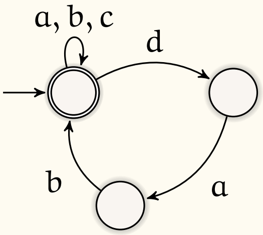
$$\forall x,\ a(x) \Rightarrow (\forall y,\ y > x \Rightarrow \neg a(y))$$

(3) Let P_2 be the property "every occurence of d must immediately be followed by an a, then a b".

a. Give a wS1S formula φ_2 equivalent to this property.

$$\forall x : d(x) \Rightarrow \big(a(x+1) \wedge b(x+1+1)\big)$$

b. Give an automaton A_2 equivalent to this property.



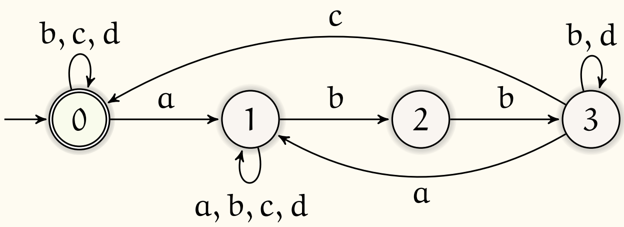
c. Give an LTL path formula ψ_2 equivalent to this property.

$$\Box\big(d \Rightarrow (\circ a \wedge \circ\circ b)\big)\ ,$$

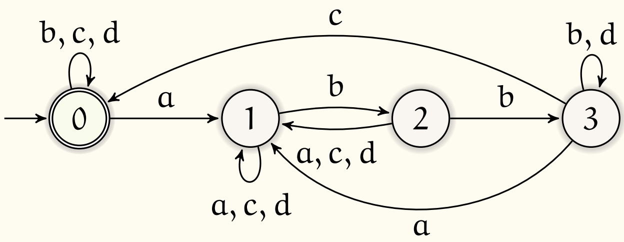
(4)

$$\Box\big(a \Rightarrow \Diamond(b \wedge \circ b \wedge \Diamond c)\big)$$

Corresponds to the NFA



or DFA



11 References

[Baier and Katoen, 2008] Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT press.
† Cited twice, pages 94 and 98.

[Büchi, 1960] Büchi, J. R. (1960). On a decision method in restricted second order arithmetic. In *Int. Congr. for Logic, Methodology and Philosophy of Science,*.
† Cited page 87.

[Clarke and Emerson, 1981] Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer.
† Cited page 92.

[Clarke et al., 1993] Clarke, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., McMillan, K. L., and Ness, L. A. (1993). Verification of the futurebus+ cache coherence protocol. In *Computer Hardware Description Languages and Their Applications*, pages 15–30. Elsevier.
† Cited page 35.

[Emerson and Halpern, 1983] Emerson, E. A. and Halpern, J. Y. (1983). "sometimes" and "not never" revisited: On branching versus linear time (preliminary report). In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, page 127–140, New York, NY, USA. Association for Computing Machinery.
† Cited page 92.

[Lowe, 1996] Lowe, G. (1996). Breaking and fixing the needham-schroeder public-key protocol using fdr. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer.
† Cited page 35.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57.
† Cited page 92.

[Staunstrup et al., 2000] Staunstrup, J., Larsen, K., Andersen, H., Hulgaard, H., Behrmann, G., Kristoffersen, K., Lind-Nielsen, J., Leerberg, H., Skou, A., and Theilgaard, N. (2000). Practical verification of embedded software. *Computer (New York)*, 33(5):68–75.
† Cited page 35.

[Thatcher and Wright, 1968] Thatcher, J. W. and Wright, J. B. (1968). Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical systems theory*, 2(1):57–81.
† Cited page 87.

[Thompson, 1984] Thompson, K. (1984). Reflections on trusting trust. https://dl.acm.org/doi/pdf/10.1145/358198.358210.

† Cited page 33.