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

January 29, 2026

I Generalities **7**

1 Meta-information about the course **8**
1.1 Note on the notes: . . . 8
1.2 Course prerequisites and student assessment . . . 8

2 Introduction: What is Formal Verification? **9**
2.1 Problem: Disaster stories . . . 9
2.2 Solution: Verification . . . 9
2.3 A Brief History of Program Proof . . . 10
2.4 Our focus in this course: Model Checking with TLA specifications . . . 11
2.5 Provisional course plan . . . 11

3 State Systems and Modelisation **11**
3.1 Brief Reminders About Nondeterministic Finite State Automata . . . 11
3.2 Modelling through automata: generalities . . . 13

4 The TLA Toolbox **13**
4.1 Linux or Windows? . . . 13
4.2 Install or download the TLA toolbox . . . 13
4.3 Install Graphviz/dot . . . 13
4.4 Dark Theme issues . . . 14

4.5 Display Scaling issues . . . 14
4.6 Annoying modification history comment . . . 14
4.7 Annoying model-checking history . . . 15

II Solving Problems via TLA$^+$ Models 16

5 The Wolf, the Goat, and the Cabbage (WGC) 19
5.1 Basic definitions / vocabulary . . . 21
5.2 `VARIABLES`: What are the system's states? . . . 22
5.3 `Init`: Where do we start? . . . 23
5.4 `Next`: How does the system move? . . . 23
5.4.1 A trivial `Next`, and parameters for the model . . . 24
5.4.2 Now for the real `Next` . . . 26
5.5 Invariants: types and other properties . . . 29
5.6 Reachability as `Target` invariant . . . 31
5.6.1 Traps of reachability as "error" traces . . . 32
5.6.2 Exploring traces . . . 33
5.6.3 Shortest trace? . . . 34

6 LAB CLASS: WGC 35

7 Indiana Jones and the Temple of Verification 36
7.1 `EXTENDS`: using / importing other modules . . . 38
7.2 Functions and records . . . 39
7.3 Debugging with `Print` . . . 39
7.4 Tightening the problem's vocabulary . . . 40
7.5 The `CHOOSE` (∋) "quantifier", weird and dangerous? . . . 40
7.5.1 Beware of impossible choices! . . . 40
7.5.2 ∋ vs. ∃: understanding non-determinism . . . 41
7.5.3 Aside on notation, history, and hidden power . . . 42
7.6 No overloading / polymorphism . . . 43
7.7 Identifying inputs / parameters of the problem . . . 43
7.8 `ASSUME`: check your inputs . . . 44

8 LAB CLASS: Indy 45

9 Semaphores, Peterson's Algorithm, and Temporal Logics 48
9.1 Semaphores . . . 49
9.2 Semaphores as a PlusCal algorithmic specification . . . 51
9.3 PlusCal in the toolbox: understanding the translation . . . 52
9.3.1 Program Control . . . 54
9.3.2 The `Spec` formula was here all along . . . 54
9.4 Invariants were Temporal (Safety) properties all along! . . . 55

9.5 Liveness Properties: poor starving semaphores! . . . 57
9.6 You can't avoid thinking about the underlying system . . . 58
9.7 Peterson's algorithm . . . 59
9.8 Safety: no problem . . . 60
9.9 What you want, you get! . . . 61
9.10 . . . but who cares what you want? . . . 62
9.11 Fat or starving, at the Scheduler's mercy . . . 63
9.12 The Temporal Logics: CTL*, CTL, & LTL . . . 63

10 The Three Islands, the Two Wolves, the Goat, and the Cabbage 65
10.1 Vocabulary and Inputs . . . 67
10.1.1 Abstraction is cutting away irrelevant details: do more for less . . . 67
10.1.2 A good abstraction often has fewer primitive concepts . . . 68
10.2 **VARIABLES**: [A → Locs] vs. [Locs → ℘ A], for you and TLC . . . 69
10.3 Fifty shades of Init: writing and altering functions . . . 71
10.3.1 TLC doesn't know what you know . . . 71
10.3.2 No convenient ↦ notation outside of records . . . 72
10.3.3 Introduction to the hidden gems ↦ (:>) and ▶(@@) . . . 72
10.3.4 The not-quite λ-expression [x ∈ S ↦ e] . . . 73
10.3.5 **IF THEN ELSE** . . . 73
10.3.6 **CASE ... OTHER** . . . 74
10.3.7 The winner: tuples and sequences . . . 74

11 LAB CLASS: Three_Islands 75

12 Scratch that recursion 77
12.1 Make a scratchpad for experimentation . . . 78
12.2 Playing with ▶ . . . 78
12.3 Reading the precedence / associativity table . . . 79
12.4 **ASSUME** as **assert**: "unit testing" . . . 80
12.5 A recursive function: fact . . . 80
12.6 Cardinality: functions vs. operators . . . 81
12.6.1 The tragedy of Darth Set-Of-All-Sets . . . 82
12.6.2 TLC vs. TLA$^+$ standard modules, **LET IN** . . . 83
12.6.3 **RECURSIVE** operators . . . 85
12.7 A straightforward sum . . . 85
12.8 **LAMBDA** and Fold / Reduce . . . 86

13 LAB CLASS: Scratchpad 87

14 LAB CLASS: The Worm, the Centipede, and the Grasshopper 88

15 LAB CLASS: Hard Mode: One Spec To Cross Them ALL! 89

16 LAB CLASS: Toggle Problems **91**

17 TLA$^+$ CheatSheet **93**

III OLD Lab classes 96

18 Preliminaries (preferably before the first lab class) **96**
18.1 Setting up a work environment . . . 96
18.1.1 Operating System . . . 96
18.1.2 Choice of Linux distribution . . . 96
18.1.2.1 Provided VM: Arch-based system . . . 97
18.1.2.2 Instructions: Arch-based system . . . 97
18.1.2.3 Instructions: Debian-based system . . . 98
18.1.2.4 Instructions: Fedora / SUSE-based system . . . 98
18.1.2.5 Instructions: Microsoft's Spyware OS . . . 98
18.1.3 Check that it works, and brush up on stuff . . . 100

19 Basic finite state systems **100**

20 Modelling complex systems using products **113**

21 Extra exercises involving rivers (from JMC's collection) **118**

22 CTL Verification **118**

IV OLD Lecture Notes: Formal Verification 120

23 Meta-information about the course **123**
23.1 Note on the notes: . . . 123
23.2 Course prerequisites and student assessment . . . 123

24 Introduction: What is Formal Verification? **124**
24.1 Problem: Disaster stories . . . 124
24.2 Solution: Verification . . . 124
24.3 A Brief History of Program Proof . . . 125
24.4 Our focus in this course: Model Checking . . . 126
24.5 Provisional course plan . . . 126

25 State Systems and Modelisation **127**
25.1 Brief Reminders About Nondeterministic Finite State Automata . . . 127
25.2 On machine: `lecture_automata_products.py` . . . 128
25.3 Modelling through automata: generalities . . . 128
25.4 Examples of Isolated Systems . . . 128

25.4.1 Digital Clock . . . 129
25.4.2 Digicode, pass 123 . . . 129
25.4.3 LIFO (Stack) and FIFO (Queue) of size 2 . . . 131
25.4.4 Incrementable Integer Variable . . . 132

26 Incrementable Unsigned Integer Variable with Overflow 133

27 Set Variable 134
27.0.1 FIFO / LIFO($\mathbf{n}, \mathbf{m}$) . . . 135
27.0.2 The Wolf, the Goat, and the Cabbage (WGC) . . . 135
27.0.3 WGC, states as functions rather than "left-bank" sets . . . 138
27.0.4 Indiana Jones and the Temple of Verification . . . 138
27.1 A Taxonomy of Automata Products . . . 140
27.1.1 Fully Synchronised Product $\otimes$ . . . 141
27.1.1.1 Fully Synchronised Product $\otimes$ for $\cap$ . . . 141
27.1.1.2 Fully Synchronised Product $\oplus$ for $\cup$ . . . 142
27.1.2 Fully Unsynchronised Product $\|$: the Shuffle . . . 142
27.1.3 Vector-Synchronised Product . . . 143
27.1.3.1 A Fully General Product . . . 145
27.1.3.2 Easy to Understand, Cumbersome to Use . . . 145
27.1.4 Named Synchronised Product . . . 146
27.1.5 Automaton Restriction . . . 147
27.2 Example Systems, Now With Some Products . . . 148
27.2.1 WGC, Now With Map-Synchronised Product . . . 148
27.2.2 Indiana Jones, now With Map-Synchronised Product . . . 151
27.2.2.1 The Solution, Concisely . . . 151
27.2.2.2 How Was I Supposed to Guess How to Handle Time? . . . 151
27.2.3 Exercise with Solution: The Bridge on the River Kwaï (FR) . . . 152
27.2.3.1 Problem Statement . . . 152
27.2.3.2 Solution . . . 153
27.2.4 Exercise with Solution: The Toggle Problems . . . 155
27.2.4.1 Problem Statement . . . 155
27.2.4.2 Solution . . . 156
27.2.5 Exercise with Solution: The Three Islands, the Two Wolves, the Goat, and the Cabbage . . . 157
27.2.5.1 Problem Statement . . . 157
27.2.5.2 Solution . . . 158
27.2.6 Exercise with Solution: Max-Weight River-Crossing Problem . . . 159
27.2.6.1 Problem Statement . . . 159
27.2.6.2 Solution . . . 160
27.2.7 Semaphores: first contact . . . 161
27.2.8 Mutable Boolean variable . . . 163
27.2.9 Sequential Programs . . . 164

27.2.9.1 Infinite Loop: `while True ...` . . . 165
27.2.9.2 `if ... else ...` . . . 166
27.2.9.3 Sequence of instructions . . . 166
27.2.9.4 Null operation: `pass` . . . 166
27.2.9.5 Application to our example . . . 166
27.2.10 Peterson's Algorithm . . . 167

28 Classical and Temporal Logics: (W)S1S, CTL*, CTL, LTL **170**
28.1 Traditional Logic on Words: (w)S1S . . . 171
28.1.1 What Does that Word Salad Even Mean? . . . 172
28.1.2 Formal Syntax and Semantics . . . 174
28.1.2.1 An Example . . . 175
28.1.3 Extending the Syntax; Writing Properties . . . 175
28.1.4 The Büchi and Thatcher–Wright Theorems . . . 178
28.1.4.1 For every NFA, there is an equivalent wS1S formula . . . 178
28.1.4.2 For every wS1S formula, there is an equivalent NFA . . . 180
28.1.5 Algorithmic Complexity and Suitability for Verification . . . 180
28.1.5.1 Reminders about Complexity Theory . . . 181
28.1.5.2 What Does it Mean for Our Purposes? . . . 182
28.2 CTL*: Computation Tree and Linear Time Logic (CTL+LTL+···) . . . 183
28.2.1 Kripke Structures and Paths . . . 184
28.2.2 Syntax and Semantics of CTL* . . . 184
28.2.3 A Few Examples, to Help the Semantics go Down . . . 186
28.2.4 LTL: Linear Time Logic . . . 188
28.2.5 CTL: Computation Tree Logic . . . 189
28.2.6 Some Useful Properties and Miscellaneous Examples . . . 190
28.2.6.1 Mutual Exclusion . . . 190
28.2.6.2 Possible Access, Liveness, "infinitely often" . . . 190
28.2.6.3 Requests Get Answers, Eventually . . . 191
28.2.6.4 No Shoes, No Service . . . 191
28.2.6.5 Interdiction . . . 191
28.2.6.6 Necessary Steps . . . 192
28.2.7 Comparing LTL, CTL, and CTL* . . . 192
28.2.7.1 Expressive Powers . . . 193
28.2.7.2 Algorithmic Complexity . . . 193
28.2.8 CTL Model-Checking Algorithm . . . 193
28.3 Exercises on Logics and Automata . . . 193
28.3.1 Exam 2020–2021 . . . 193
28.3.1.1 Problem Statement . . . 193
28.3.1.2 Solution . . . 194

Part I

Generalities

1 Meta-information about the course **8**
1.1 Note on the notes: . . . 8
1.2 Course prerequisites and student assessment . . . 8

2 Introduction: What is Formal Verification? **9**
2.1 Problem: Disaster stories . . . 9
2.2 Solution: Verification . . . 9
2.3 A Brief History of Program Proof . . . 10
2.4 Our focus in this course: Model Checking with TLA specifications . . . 11
2.5 Provisional course plan . . . 11

3 State Systems and Modelisation **11**
3.1 Brief Reminders About Nondeterministic Finite State Automata . . . 11
3.2 Modelling through automata: generalities . . . 13

4 The TLA Toolbox **13**
4.1 Linux or Windows? . . . 13
4.2 Install or download the TLA toolbox . . . 13
4.3 Install Graphviz/dot . . . 13
4.4 Dark Theme issues . . . 14
4.5 Display Scaling issues . . . 14
4.6 Annoying modification history comment . . . 14
4.7 Annoying model-checking history . . . 15

1 Meta-information about the course

1.1 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.

For the TLA$^+$-related parts, I used *A Science of Concurrent Programs* and *Specifying Systems*, by Leslie Lamport, as reference documents.

Any mistakes in this document are, likely, mine.

1.2 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.
- ◇ First-order logic.

Assessment: final examination, on paper.

2 Introduction: What is Formal Verification?

2.1 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) . . .

2.2 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. ...

2.3 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.

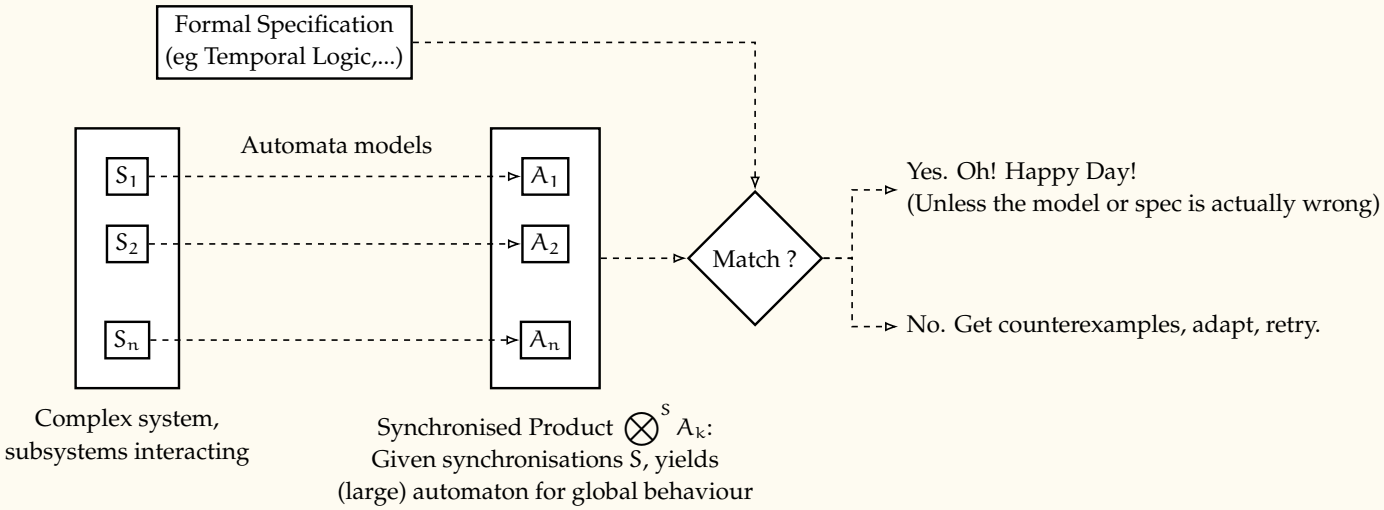
(6) 1980-1990 Nombreux résultats théoriques

(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

2.4 Our focus in this course: Model Checking with TLA specifications



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.

2.5 Provisional course plan

TODO

3 State Systems and Modelisation

3.1 Brief Reminders About Nondeterministic Finite State Automata

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

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

Sémantique: langage reconnu par un automate:

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

Déterminisme, non-déterminisme et ε

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

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)

I have a complete implementation of NFAs on https://github.com/vincent-hugot/nfalib, but we'll probably not need it for the TLA version of this course.

3.2 Modelling through automata: generalities

A shift on philosophy regarding automata:

3A: 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.

4 The TLA Toolbox

4.1 Linux or Windows?

Linux, of course, on general principles. That said, both pieces of software we'll use, the Toolbox and Graphviz/dot, exist on Windows. I won't be able (or willing) to help you with any Windows-specific problem. MacOS is also a thing that exists, I'm told.

4.2 Install or download the TLA toolbox

(1) Arch Linux (AUR): `yay -S tla-toolbox`.
I am the maintainer of that package, so it should be up-to-date. Complain if not.

(2) Everything else:
`https://github.com/tlaplus/tlaplus/releases/latest`
Snaps / Flatpacks etc probably won't be terribly up-to-date, I advise against using them.

4.3 Install Graphviz/dot

Graphviz is graph-visualisation software, which the toolbox uses to... well... *visualise*... the *graph*... of the state machines we specify. Makes sense. Use your package manager:

(1) Arch: `sudo pacman -S graphviz`

(2) Ubuntu: `sudo apt install graphviz`

(3) Fedora: `sudo dnf install graphviz`

(4) Whatever else: `https://graphviz.org/download/`

If you get an error when the toolbox tries to render the graph (we'll get there in the first lesson), where it says it can't find the dot executable, find the **absolute path** where the dot executable is with `$ whereis dot`, for instance `/usr/bin/dot`, and paste that in *Preferences* / *PDF Viewer* (you can type filter text in the upper left corner to find the right page fast) / *Specify dot command*. Do not putter around with PATH, the toolbox may or may not have what you expect in its environment variables; using an absolute path will always work.

4.4 Dark Theme issues

If your machine uses a **dark theme**, the toolbox will tend to be an unreadable mess. You might want to either **(1)** switch to a light theme machine-wide, or **(2)** explicitly switch the toolbox to its dark theme; — the only downside is that white-on-pink error traces are unreadable, and I can't find the setting for this font style; if you find it, let me know, — or **(3)** find a way to launch the toolbox in full light theme anyway (the `-cssTheme none` option has been reported to work, but I couldn't validate that on my machine — KDE).

4.5 Display Scaling issues

Display scaling other than 100% can also create issues. While there are command lines on the internet that purport to solve this, such as

```
bash -c 'env GDK_DPI_SCALE=$(awk -v a="${GDK_SCALE:-1}"
       -v b="${GDK_DPI_SCALE:-1}" "BEGIN{print (a*b)}")
     GDK_SCALE="${GDK_SCALE:+1}" <path to toolbox> "%f"'
```

none of them solved the problem for me on KDE. If you find solutions, let me know.

4.6 Annoying modification history comment

By default, when creating a new specification file, the toolbox writes a comment at the bottom which it updates each time you modify the TLA file:

```
\* Modification History
\* Last modified Sat Jan 11 12:21:35 UT 2024 by YOU
...
```

It is quite silly and redundant with git, because of course anything resembling code should always be in a git, even if just a local one (you can afford a `git init .`). You can either remove the comment altogether, it won't come back, or disable the behaviour altogether in *Preferences* / *Module editor* / *Add a modification history to new spec*.

4.7 Annoying model-checking history

Each time you check a model, a snapshot of the result is stored as files, to a maximum of the N latest, N = 10 by default. This is rarely useful for us, and clutters the view and our work folders.

Go to *Preferences* / *TLC Model Checker* / *Number of model snapshots to keep*, and set to 1.

Part II

Solving Problems via TLA$^+$ Models

5 The Wolf, the Goat, and the Cabbage (WGC) 19
- 5.1 Basic definitions / vocabulary . . . 21
- 5.2 `VARIABLES`: What are the system's states? . . . 22
- 5.3 `Init`: Where do we start? . . . 23
- 5.4 `Next`: How does the system move? . . . 23
  - 5.4.1 A trivial `Next`, and parameters for the model . . . 24
  - 5.4.2 Now for the real `Next` . . . 26
- 5.5 Invariants: types and other properties . . . 29
- 5.6 Reachability as `Target` invariant . . . 31
  - 5.6.1 Traps of reachability as "error" traces . . . 32
  - 5.6.2 Exploring traces . . . 33
  - 5.6.3 Shortest trace? . . . 34

6 LAB CLASS: WGC 35

7 Indiana Jones and the Temple of Verification 36
- 7.1 `EXTENDS`: using / importing other modules . . . 38
- 7.2 Functions and records . . . 39
- 7.3 Debugging with `Print` . . . 39
- 7.4 Tightening the problem's vocabulary . . . 40
- 7.5 The `CHOOSE` (∋) "quantifier", weird and dangerous? . . . 40
  - 7.5.1 Beware of impossible choices! . . . 40
  - 7.5.2 ∋ vs. ∃: understanding non-determinism . . . 41
  - 7.5.3 Aside on notation, history, and hidden power . . . 42
- 7.6 No overloading / polymorphism . . . 43
- 7.7 Identifying inputs / parameters of the problem . . . 43
- 7.8 `ASSUME`: check your inputs . . . 44

8 LAB CLASS: Indy 45

9 Semaphores, Peterson's Algorithm, and Temporal Logics 48
- 9.1 Semaphores . . . 49
- 9.2 Semaphores as a PlusCal algorithmic specification . . . 51
- 9.3 PlusCal in the toolbox: understanding the translation . . . 52
  - 9.3.1 Program Control . . . 54
  - 9.3.2 The `Spec` formula was here all along . . . 54
- 9.4 Invariants were Temporal (Safety) properties all along! . . . 55
- 9.5 Liveness Properties: poor starving semaphores! . . . 57

9.6 You can't avoid thinking about the underlying system . . . 58
9.7 Peterson's algorithm . . . 59
9.8 Safety: no problem . . . 60
9.9 What you want, you get! . . . 61
9.10 . . . but who cares what you want? . . . 62
9.11 Fat or starving, at the Scheduler's mercy . . . 63
9.12 The Temporal Logics: CTL*, CTL, & LTL . . . 63

10 The Three Islands, the Two Wolves, the Goat, and the Cabbage 65
10.1 Vocabulary and Inputs . . . 67
10.1.1 Abstraction is cutting away irrelevant details: do more for less . . . 67
10.1.2 A good abstraction often has fewer primitive concepts . . . 68
10.2 **VARIABLES**: `[A → Locs]` vs. `[Locs → ℘ A]`, for you and TLC . . . 69
10.3 Fifty shades of `Init`: writing and altering functions . . . 71
10.3.1 TLC doesn't know what you know . . . 71
10.3.2 No convenient ↦ notation outside of records . . . 72
10.3.3 Introduction to the hidden gems ↦ (:>) and ►(@@) . . . 72
10.3.4 The not-quite λ-expression $[x \in S \mapsto e]$ . . . 73
10.3.5 **IF THEN ELSE** . . . 73
10.3.6 **CASE ... OTHER** . . . 74
10.3.7 The winner: tuples and sequences . . . 74

11 LAB CLASS: Three_Islands 75

12 Scratch that recursion 77
12.1 Make a scratchpad for experimentation . . . 78
12.2 Playing with ► . . . 78
12.3 Reading the precedence / associativity table . . . 79
12.4 **ASSUME** as **assert**: "unit testing" . . . 80
12.5 A recursive function: `fact` . . . 80
12.6 Cardinality: functions vs. operators . . . 81
12.6.1 The tragedy of Darth Set-Of-All-Sets . . . 82
12.6.2 TLC vs. TLA$^+$ standard modules, **LET IN** . . . 83
12.6.3 **RECURSIVE** operators . . . 85
12.7 A straightforward `sum` . . . 85
12.8 **LAMBDA** and Fold / Reduce . . . 86

13 LAB CLASS: Scratchpad 87

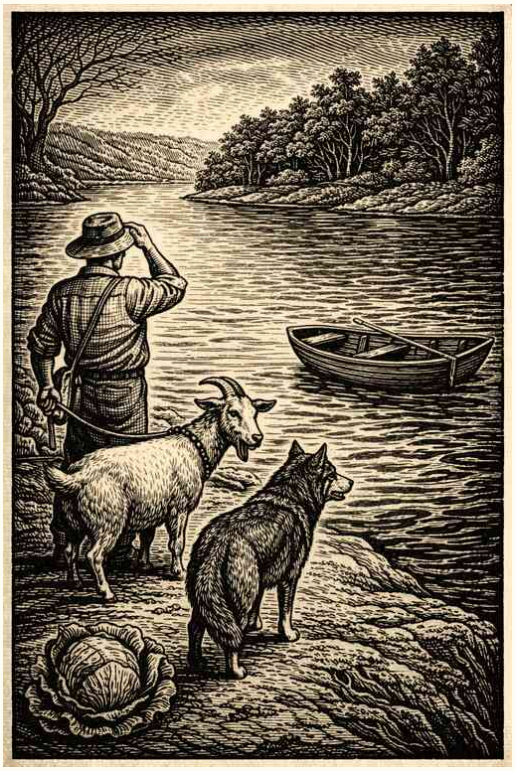
14 LAB CLASS: The Worm, the Centipede, and the Grasshopper 88

15 LAB CLASS: Hard Mode: One Spec To Cross Them ALL! 89

16 LAB CLASS: Toggle Problems **91**

17 TLA$^+$ CheatSheet **93**

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



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

> *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.

Let us solve that by writing a specification capturing 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.

[(a)] . . . and all the illustrations in the style of wood engravings are shamelessly obtained from generative AI tools. That is, in fact, the sum total of AI use that is permissible in this course :-)

5.1 Basic definitions / vocabulary . . . 21
5.2 **VARIABLES**: What are the system's states? . . . 22
5.3 `Init`: Where do we start? . . . 23
5.4 `Next`: How does the system move? . . . 23
5.4.1 A trivial `Next`, and parameters for the model . . . 24
5.4.2 Now for the real `Next` . . . 26
5.5 Invariants: types and other properties . . . 29
5.6 Reachability as `Target` invariant . . . 31
5.6.1 Traps of reachability as "error" traces . . . 32
5.6.2 Exploring traces . . . 33
5.6.3 Shortest trace? . . . 34

5.1 Basic definitions / vocabulary

In order to model the problem, we need to have a few basic definitions for the "vocabulary" of the problem, so we can describe what's going on. Here, we have a few colourful characters / actors. Let's first give them names, then we'll be able to describe their movements...

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.

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

Let's create a new TLA$^+$ specification in the toolbox. Let's call it `WGC.tla`.

The empty specification looks like this,

```
---------------------- MODULE WGC ----------------------


=========================================================
```

and should show as "parsed" in green in the bottom-right corner, meaning that there are no syntax errors so far. Everything goes between those two lines. Anything outside is treated as a comment, essentially.

Let's begin:

```
W ≜ "Wolf"
G ≜ "Goat"
C ≜ "Cabbage"
F ≜ "Farmer"

A ≜ {W,G,C,F}
```

Note about syntax: *TLA$^+$ is mathematics, so it is rendered as such in documents; however, it is coded in ASCII, like LaTeX, with which it even shares some points of syntax. The $\triangleq$ symbol is written == in code, and means "equals by definition". Look in Sec. 17[p93]: "TLA$^+$ CheatSheet" for the ASCII equivalents of TLA$^+$ syntax. Additionally, whenever I use Greek letters or other notations in the maths, I'll render the corresponding TLA$^+$ identifiers in the same way in this document, but they'll have ASCII names in the code. For instance, in the next exercise we'll have τ and $\overline{\tau}$.*

Those are *operator* definitions. In that case the operators have no parameters, they are constants. Operators in TLA$^+$ are *a bit like* values and functions in programming languages, but there are not first-class citizens. There is no "operator" type, they are a syntactic construct of the language, that's all.

Operators evaluate to values, with the following types:

Primitive types: booleans, integers, strings, and model values (we won't deal with the latter).

Complex types: sets, sequences, records, and functions (which shouldn't be confused with operators; more on that later).

In that case, we have defined five nullary operators of types string and set of strings, respectively.

5.2 VARIABLES: What are the system's states?

Now we have words to speak of our different actors. What do we need to keep track of as the system moves? 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. Note that any actor that is not on the left is necessarily on the right.[b]

```
VARIABLES left (* left ∈ ℘ A: set of actors on the left bank *)
```

VARIABLES[c] is a TLA$^+$ keyword declaring the *variables*, or *flexible variables* of the system. Technically there is also a notion of *rigid variables*, denoted by **CONSTANTS**, but we won't use it in this course — and even if we did, I'd call them *constants*. So I'll just speak of *variables*.

A state of the system is an assignment of values to the variables. Here, there is only one variable, `left`, which can take all the values in $\wp(A)$[d], so that's our states space.

Note that TLA$^+$ is untyped, meaning that aside from the comment which, being a comment, has no action, there is nothing preventing `left` from taking other values. We'll see later how we enforce a variable's type.

Even within the states space $\wp(A)$, that does not mean all those states are actually *reachable* is we follow the rules of the game: start on the left, avoid conflicts, etc. Whether we can

[b] 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

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

establishes a bijection between predicates on a set S and subsets of S.

[c] **VARIABLE** is also a keyword and means exactly the same thing. I personally always use the plural version.

[d] This is the **powerset** of A, or the set of all subsets of A. By definition $s \in \wp(A) \Leftrightarrow s \subseteq A$.

There are many notations for it, all but one of them a stylised letter P: $P, \mathcal{P}, \mathbb{P}, \ldots$ Mine is called the "Weierstraß P", I try to stick to it. The other notation is 2^S, which is good for cardinality computations: $\left|2^S\right| = 2^{|S|}$. This specialises the $B^A = (A \to B)$ notation for sets of functions when $|B| = 2$.

actually reach a state where everyone is safely on the right bank is precisely the question we are doing all that to solve.

Let us note in passing that we therefore have at most $|\wp(A)| = 2^{|A|} = 2^4 = 16$ distinct possible states for our system.

5.3 `Init`: Where do we start?

We now know what states *look like*, and we have the vocabulary to describe them. So let us describe what the *initial states* are.

Here we have just one: everybody is on the left bank, at the start. Let us call `Init` the *initial predicate*. `Init` is not a reserved word of TLA$^+$, but in practice we shall always use that convention.

`Init`, the *initial predicate*, is a *state predicate*, which basically means a predicate of first-order logic (in TLA$^+$ syntax), speaking of the system variables or constants, whose models (that is to say, assignment of the variables that make the predicate true) are exactly the states that model the initial configurations / states of our system.

Here, we start with everyone on the left bank. So, remembering what the variable `left` represents, we have:

```
(***************************************************************************)
(* INITIAL STATE                                                           *)
(***************************************************************************)

Init ≜   (* everyone starts on left bank *)
    left = A
```

Note that $\triangleq$ is the "*definition* =" whereas = is the usual "equality test" in maths.

5.4 `Next`: How does the system move?

Now we get to the difficult part. Specifying the system's *transitions*. That is, how it moves, possibly non-deterministically, from a given state to the next.

We call a pair of successive states a *step*.

We specify the system transitions by giving a *next-state relation*, or an *action / action predicate*, that is to say a predicate which is true or false of a step. Its models must be exactly the valid steps of the system.

Syntactically, it is exactly the same as a state predicate like `Init`, except in that it can , additionally, contain *primed variables*. Indeed, each variable (here we just have `left`) has a

primed counterpart (here `left'`) which simply means represents the value of the variable in the next state.

By the same convention as for `Init`, the next-state relation will always be called `Next`.

5.4.1 A trivial `Next`, and parameters for the model

Before we get into the weeds of solving our problem, let's begin by giving, temporarily, a trivial value for next, so that we can validate that our tools actually work on something simple. *Then* we can write a more complex specification.

For now, let's say that our system loops infinitely, never changing state.

```
(***************************************************************************)
(* TRANSITIONS                                                             *)
(***************************************************************************)

Next ≜
    left' = left
```

Technical detail: *for technical reasons, `left = left'`, while mathematically equivalent to `left' = left` and completely correct TLA$^+$, would not work in practice with the model-checking tool, TLC. Just remember to put the primed variables on the left and you'll be fine.*

Let's create a new "model" in the toolbox. It will be `Model_1`. The file

`WGC.toolbox/WGC___Model_1.launch`

thus created should probably be added to your git.

In *Model Overview / what is the behaviour spec?*, select "Initial predicate and next-state relation", and fill the fields with `Init` and `Next`, respectively.

Run the model (F11), you should not see any error.

In *TLC options* (revealed by *Model overview / What to check? / additional TLC options*) / *Features*, check *Visualize state graph after completion of model-checking*. You will need the Graphviz/dot software installed for this to work; see Sec. 4.3[p13]: "Install Graphviz/dot".

Run the model again, now you should see a "State Graph" tab.

Note: *the toolbox's internal PDF viewer can be hard to wrangle when the graph gets large (or, rather, "not tiny"). The generated PDF is to be found in*

`WGC.toolbox/Model_1/Model_1.pdf`

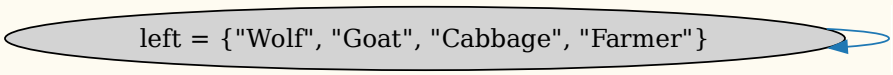
so you can just open it with a PDF viewer like Okular or whatever it is you use, and it will automatically[(e)] *refresh the view when it is regenerated. Very convenient if you tile the windows, or move the graph to a second monitor.*

[(e)] Only on Linux; on Windows Foxit and Adobe Reader will need you to refresh manually.

It should contain a single state

$$\texttt{left} = \{\texttt{"Wolf"}, \texttt{"Goat"}, \texttt{"Cabbage"}, \texttt{"Farmer"}\}$$

looping onto itself.



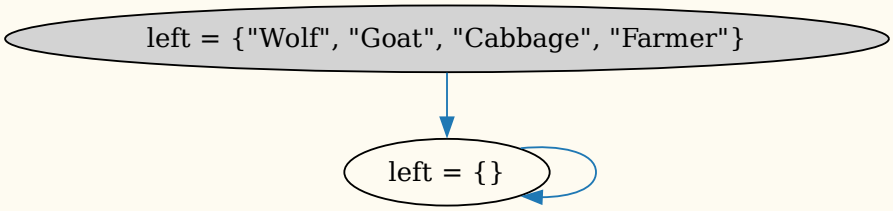
This is indeed the system we have described.

Now try with

```
Next ≜
    left' = ∅
```

This is saying that a step is valid if and only if the next state is "there is nobody left".

Run the model. Now you should see two states, with the second one, where nobody is left, looping onto itself.



Now try again with

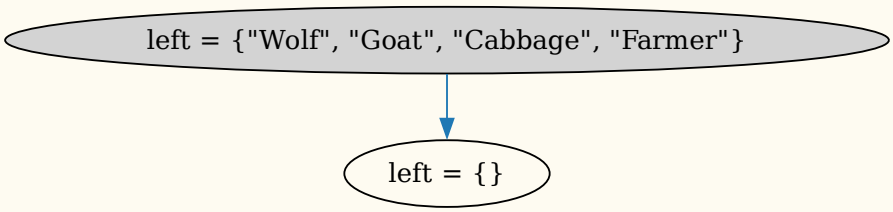
```
Next ≜
    left ≠ ∅ ∧ left' = ∅
```

This is saying that a step is valid if and only if there is somebody left and the next state is "there is nobody left".

Run the model. This time you get an error. An error at checking time means "the system does behave in a way it should *not*". This is bit surprising, given that we have only specified (a tiny bit of) how the system *behaves*, and nothing (yet) of how it *should behave*.

It turns out that, by default, the toolbox checks against a behaviour called "deadlock". Look at the TLC error message: "Deadlock reached", it says.

Look at the graph.



It's the same as before, but there is no loop on the second state, now. Indeed, by `Next`, there is no valid step from that state. That's what "Deadlock" means for TLA$^+$. You're stuck in a state with no valid move at all, not even to itself.

Whether we care about such a notion of deadlock, and actually consider that an error, depends entirely on what we are modelling and what we're trying to achieve.

For the current problem, we morally don't give a damn about deadlocks, because our objective is to determine whether we can reach a state where everyone is on the right bank. So long as we get there, we don't care if we get stuck in that target state, or if other strategies would have gotten us stuck on the way there.

If the toolbox annoys you with this, simply uncheck "Deadlock" in *Model Overview/What to check?*.

5.4.2 Now for the real `Next`

We have validated that our tools work as expected, now let's start actually modelling our system's transitions.

As always, it's a bad idea to do too many things at the same time, so let's break down the concept of valid step into smaller, bite-sized morsels.

We have a notion of *move* whereby the Farmer can move himself and, optionally, at most one other actor, either left-to-right or right-to-left. Furthermore, the move must be *licit* according to the "don't let them eat each other" constraints.

Let's deal with those things separately. Now, when it comes to moving, left-to-right and right-to-left seem to be basically the same notion with source and target reversed, so we could and probably should abstract that to avoid duplicating logic, but it's our first time, and we didn't actually pick the simplest way to encode states if we wanted to do that, so, let's not.

So, we have three notions: *move left-to-right*, *move right-to-left*, and *licit*.

Let's code *one*, and test it.

Moving left-to-right means that

(1) The Farmer is left, and

(2) possibly, another actor a that is left is selected, and

(3) both go to the right: $\texttt{left}' = \texttt{left} \setminus \{F, a\}$

Superficially there are two cases: either the Farmer travels alone, or not, but if we allow a to possibly be the Farmer himself, we don't need to explicitly distinguish the cases: $\{F, F\} = \{F\}$, that's how sets work.

Writing that in first-order logic, and therefore in TLA$^+$, is straightforward, using $\exists$:

```
mvlr ≜  (* move from left to right *)
    ∧ F ∈ left
    ∧ ∃ a ∈ left : left' = left \ {F,a}

Next ≜ mvlr
```

Syntax note: *to help write complex formulæ without drowning in parentheses, TLA$^+$ allows and whitespace/indentation-sensitive "bulleted-list" style of handling big conjunctions and disjunctions:*

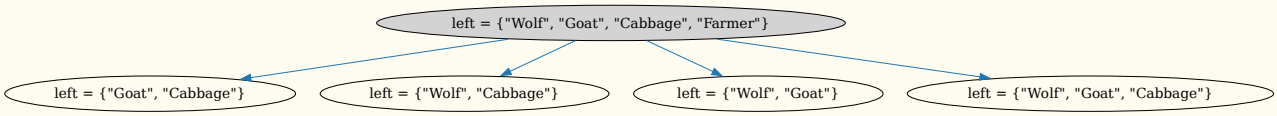
```
Op ≜  ∧ x
      ∧  ∨ y
         ∨ z
```

means the same thing as

```
Op ≜ x ∧ ( y ∨ z)
```

The ∧/∨ must line up vertically.

Run the model, you will now find five states:

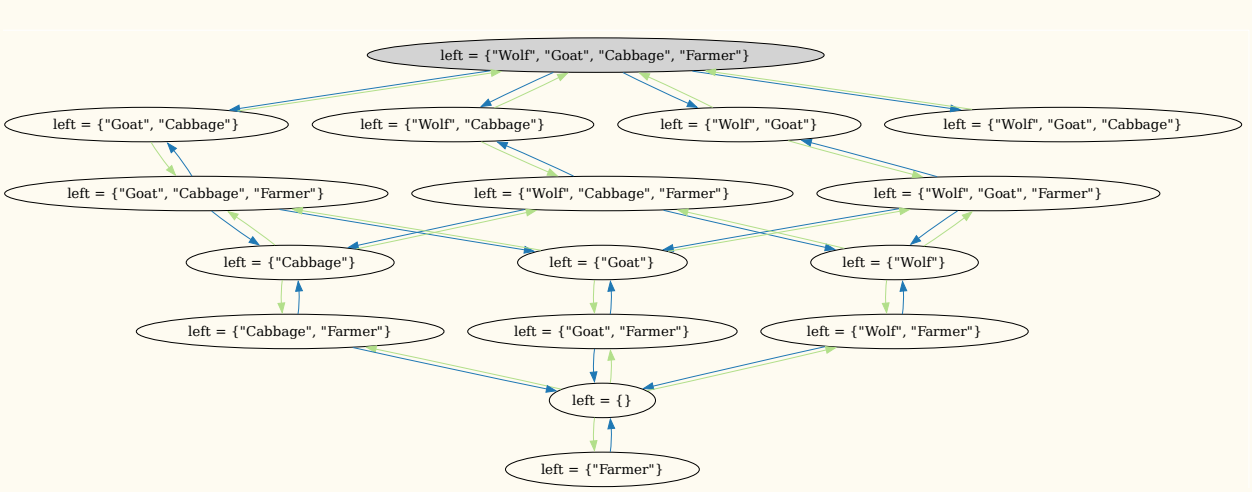


our initial, leading to three where the farmer went right with each of WGC, and one where he went right on his own. And then, in all cases, everyone is stuck, which is as expected as nobody can move left-to-right anymore — the Farmer is missing — and nobody can move right-to-left, as. . . we have not specified that yet.

Let's do it:

```
mvrl ≜  (* move from right to left *)
    ∧ F ∈ A \ left
    ∧ ∃ a ∈ A \ left : left' = left ∪ {F,a}

Next ≜ mvlr ∨ mvrl
```

Running that, we find that we now have a rather complicated state machine graph with 16 states. In other words, all possible states are reached.



Toolbox note: *Since* `Next` *is a pure disjunction, meaning, morally, that there are different kinds of transitions, the graph generated by the toolbox helpfully colours the transitions according to which clause "created" them.*

This makes perfect sense, because at this point we have modelled moving *without restriction*. The Farmer can ferry any actor to the right, come back, and do it again as many times as needed, then either stay or go back alone. Thus any state is reachable.

That checks out, let's move on to the nub of the problem: the restrictions.

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 on the bank under consideration, then the Farmer must be with them. Let $s \subseteq A$ be a collection of actors; they are licit on the same bank if

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

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

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

Let us note in passing that our initial state is licit. If that weren't the case, the problem would have no solution. We shall come back to that shortly.

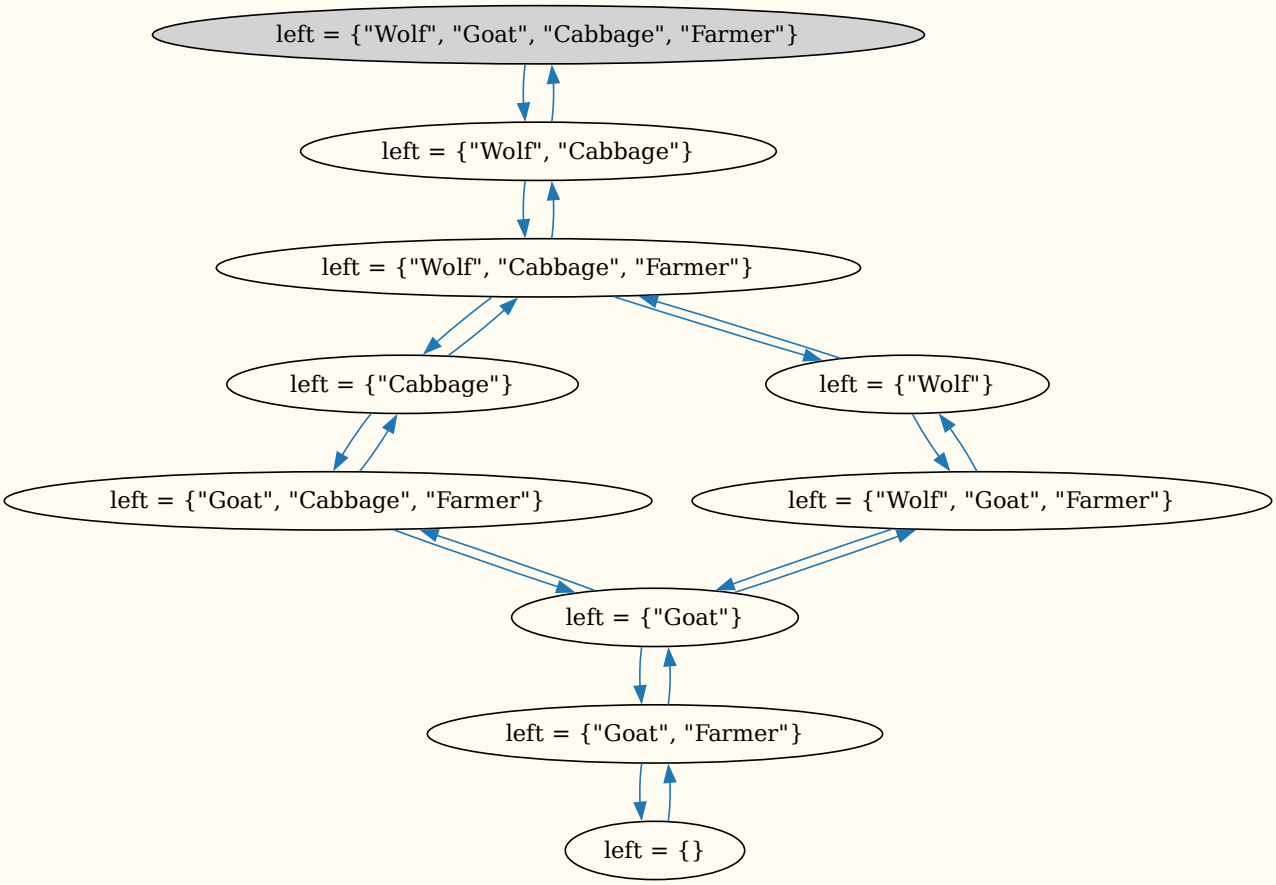
Let us translate the maths we just wrote in TLA$^+$. Note that, for the first time, we write operators with a parameter:

```
l0(s) ≜  (* Set s ⊆ A licit on a bank: no conflict *)
    ( { W, G } ⊆ s  ∨  { G, C } ⊆ s ) ⇒ F ∈ s
l(s) ≜   (* Set s licit assignment for left: no conflict on either bank *)
    l0(s)  ∧  l0(A \ s)
```

Now we just have to restrict our moves: we move only if the next state is licit.

```
Next ≜   (* move, and target state is licit *)
    ∧ mvlr ∨ mvrl
    ∧ l(left')
```

Run the model.



Now we have ten states, culminating in the state `left` = ∅, which means we have indeed crossed the river successfully under those constraints. Reading the graph, the idea is to isolate the Goat — which makes sense given that the Goat conflicts with both the others.

At that point we have, intuitively, answered the question, but we're not quite done yet. We can't always (or often) just look at the graph and conclude. We have formally specified *how the system behaves*, there remains to specify *what we expect of it*, how it *should* behave, and let the toolbox conclude whether the two match.

5.5 Invariants: types and other properties

An *invariant* is a state predicate that is, or rather must be, true for all states of the system.[f] We are going to write invariants to express what we expect of our system.

[f] A more precise definition is

> *An invariant* Inv *of a specification* Spec *is a state predicate such that* Spec ⇒ □Inv *is a theorem.*

Recall that we wrote in a comment that `left` $\subseteq$ `A`, but didn't enforce it. Now is the time. That kind of property is a *typing invariant*.

Another thing we would like to check is that every state is licit. Let's write that in our invariant, which we'll call `Verif` for now:

```
(***************************************************************************)
(* INVARIANTS                                                              *)
(***************************************************************************)

Verif ≜
    ∧ left ⊆ A   (* typing *)
    ∧ l(left)   (* every state is licit *)
```

Note: *Unlike* `Init` *and* `Next`, `Verif` *is not a naming convention we'll follow in the future, which is why I didn't set syntax highlighting for it. Instead we'll have separate invariants for typing the variables (*`Type`*) and for more interesting properties (*`Inv`*). But that's for the next exercise.*

In order for `Verif` to actually be checked by the tools, we need to go to *Model overview/What to check?/Invariants* and add `Verif` (and make sure it is checked).

Now we run the model. There shouldn't be any error. Let's make sure our invariant actually *does* something, though. Let's add to it

```
    ∧ ⊥
```

Now if you run the model and *don't* get an error (on the initial state in that case), it means the invariant is not actually checked.

Always check that what you write actually does something. There are plenty of ways to press a button and get the tools to tell you "everything is fine", because nothing actually happens. Not having the invariant activated is one way. Having a system that doesn't actually do anything interesting is another. For instance, having

```
Next ≜ left' = left
```

will still satisfy `Verif`. If you have no states, none of them run a risk of violating your carefully crafted invariants. This is why you shouldn't just verify that "the system never does anything bad" (which is what invariants do), but also that "the system actually *does* something useful at some point". We'll come back to this shortly.

Let's think again about the `∧ l(left)` part of `Verif`. Technically we already checked that all states are licit when we generated them in `Next`:

```
Next ≜   (* move, and target state is licit *)
    ∧ mvlr ∨ mvrl
    ∧ l(left') (* here is where we checked... *)
```

The □ is read "always". We'll come back to that when we study temporal logic.

We could say it's redundant to put it here as well, and that it means we'll check the same states twice and waste a little bit of energy. . . and that's *technically* true but that's not at all the right way of thinking.

First, we actually never mechanically checked the initial state.

Second, if we ever modify `Next`, we might accidentally remove or deactivate the *licit* test without realising it.

Third, morally, once **VARIABLES** are decided upon, the specification of the system and its invariants should be independent. In practice, they might be written by different people or teams.

Fourth, redundancy is not always a bad thing. We don't do formal verification with the goal of minimising the execution time of the verification itself. We do it to develop confidence that our system is correctly designed. If we want confidence that all states are licit, then we want to see that property *explicitly* checked and listed as a theorem by our verification tools.

We shouldn't have to read `Init` and `Next` and write an inductive proof in our head, no matter how trivial, to check our theorem. We might get it wrong, and *doing that for us is the reason the bloody verification tools exist*. Let's use them.

5.6 Reachability as `Target` invariant

Now, let's get back to checking that "the system actually *does* something useful at some point".

Our question is "does the system reach a state where everybody is on the right bank?". In other words, can we reach a state such that `left = ∅`?

This is not an invariant. We do not want to check that "every state is `left = ∅`". Nevertheless, we are going to do this with invariants. Recall the duality of $\exists$ and $\forall$:

$$\exists x \in X : P(x) \quad \Longleftrightarrow \quad \neg \forall x \in X : \neg P(x)$$

Checking an invariant I is checking "$\forall s \in$ `states` : I". So if we want to check whether the system has a state with property P, we can just test for an invariant $\neg$P. If the invariant is violated, we win; if not, we lose.

Let's set up $\neg$`Target` as an invariant in *What to check?*, and write a state predicate `Target` specifying our target state:

```
Target ≜ (* ¬Target as invariant ⇒ show how to reach Target *)
    left = ∅ (* reach state where nobody is left? *)
```

Run the model. You will get an error: "Invariant $\neg$`Target` is violated." **This is good!** This is what we expected. It means that there exists, and we have reached, a `Target` state.

Furthermore, we see an "error" trace, here really a *success* trace, of length 8, leading to a the target state. That's our final answer to the question we asked. *In fine*, our specification, and the result of the model-checker, constitute a proof[g] that yes, we can reach the right bank under those conditions. We're *almost* done.

5.6.1 Traps of reachability as "error" traces

There is a tiny trap, however. We know that the target has been reached, but does that execution of the model-checker, in and of itself, also prove that `Verif` is satisfied? If not, we have a big problem.

TLC stops as soon as it encounters (what it considers to be) an error; did it actually check `Verif` before stopping? Actually, it depends.

To show that, let's add

```
∧ left ≠ ∅
```

to `Verif`. Now we know that `Verif` does *not* hold. If TLC tells us "victory! target is reached!" without pointing out that, actually, that's utterly meaningless because `Verif` is violated, we have a problem.

We run the model, get "Invariant Verif is violated." Sigh of relief.

But wait... does that depend on the order in which the invariants are tested? What if we had written `¬Target` first and `Verif` later? Let's test that, by adding `Verif` *again* as an invariant, and unchecking the first. You should have

```
[ ] Verif
[v] ¬Target
[v] Verif
```

Run the model. "Invariant `¬Target` is violated."

Crap.

It *is* dependant upon the order in which the invariants are evaluated. Are we certain that they are always evaluated in the order given in the model? We could check how TLC is implemented and everything but really *it doesn't matter*, because this is the kind of finicky implementation detail which we absolutely do not want to rely on or even think about. Doing so would completely undermine the confidence which we are trying achieve by doing formal verification in the first place.

Let's stick to clear and simple rules: **if you get an error in an invariant, you can't assume that the other invariants hold**. Plain and simple.

[g] ... to the extent that we correctly specified the system, and that the tools we used are correctly implemented, etc.

So what do we do: first we check `Verif` *without* `¬Target`, then we check again *with* `¬Target`.

It does not hurt to *also* put true invariants *before* fake negative target "invariants", because we're *pretty sure* they're evaluated in order, and that this won't change with the next version of TLC, it's just not the kind of thing we should *rely* on. Ever.

Of course, there is also a much cleaner way to specify that a target is reached, which we shall see later (liveness temporal formulæ, ◇ modality).

We'll still use this one because **(1)** it's conceptually simpler, and **(2)** it actually gives you a trace to the target, instead of just saying "OK, it works!". Which is neat.

5.6.2 Exploring traces

Speaking of traces. We get

```
[
left ↦ {"Wolf", "Goat", "Cabbage", "Farmer"}
],
[
left ↦ {"Wolf", "Cabbage"}
],
...
[
left ↦ ∅
]
```

which neat, yes, but not super-easy to read, because we would like to actually see who is on the right bank. Doing mental set complements all the time is a bit exhausting.

Fortunately, we don't need to add a redundant variable *right* to the specification itself: we can add expressions to *Error-Trace Exploration*.

Add `right` ≜ `A \ left` and click *Explore*. Now we see:

```
[
 left ↦ {"Wolf", "Goat", "Cabbage", "Farmer"},
 right ↦ ∅
],
[
 left ↦ {"Wolf", "Cabbage"},
 right ↦ {"Goat", "Farmer"}
],
...
```

Much better.

Too bad we can't — to my knowledge — do the same for the state graph.

Weirdness Alert: there is a difference in behaviour between named and unnamed trace expressions *when they contain primed variables*. You can see that by adding both `left'` and `next` $\triangleq$ `left'`. It's occasionally useful when you want access to values in the *previous* state, but I do not actually have a principled explanation for why it behaves that way. If you know or find one, please share and get a +1 to your final mark :-)

5.6.3 Shortest trace?

So, we found a success trace in which we have full confidence, and we have the tools to read and understand it.

But is it the shortest? That is to say, is there potentially a way to get to the right bank in fewer moves?

A quick search of the documentation reveals

> *[TLC's] default method [...] is to find the graph of all reachable states using breadth-first search. This has the advantage that if TLC finds a violation of a safety property, then it will produce a shortest possible behavior that exhibits the error.*

Surely that means that if you get an error (/success) trace, it is the shortest. After all, the documentation states that most unambiguously. Right? Right? RIGHT?

WRONG!

TLC can and does by default (via the toolbox or `-workers=auto` setting) use several worker threads: one per core of your CPU. It's 2026, you probably have several cores. I have 12 on my main machine.

That means that there are several, in my case 12, instances of TLC exploring the states space in parallel; the instant one of them says "I found an error trace!", everything stops. It is quite possible for Worker 3 to find a trace of length 15 *before* Worker 11 finds a trace of length 14. Maybe Worker 11 had a lot more states to go through, or maybe there was something else running on the same core as Worker 11 that slowed it down. Who knows. And next time you run it, maybe you'll get a trace of length 13. Who knows.

For our current WGC problem, we can convince ourselves by looking at the state graph that it can't happen. (We have infinite possible success traces, but they are all extensions of the two len-8 shortest traces.) For more complex problems, it can and definitely will happen. Sometimes.

Again, let's stick to simple rules: **if you want a *guaranteed* shortest trace, you *must* set *workers* to 1 in TLC options.**

6 LAB CLASS: WGC

Now take your machine, and go through the WGC exercise on your own, making sure you understand all the points developed in the previous section and can find your way in the toolbox.

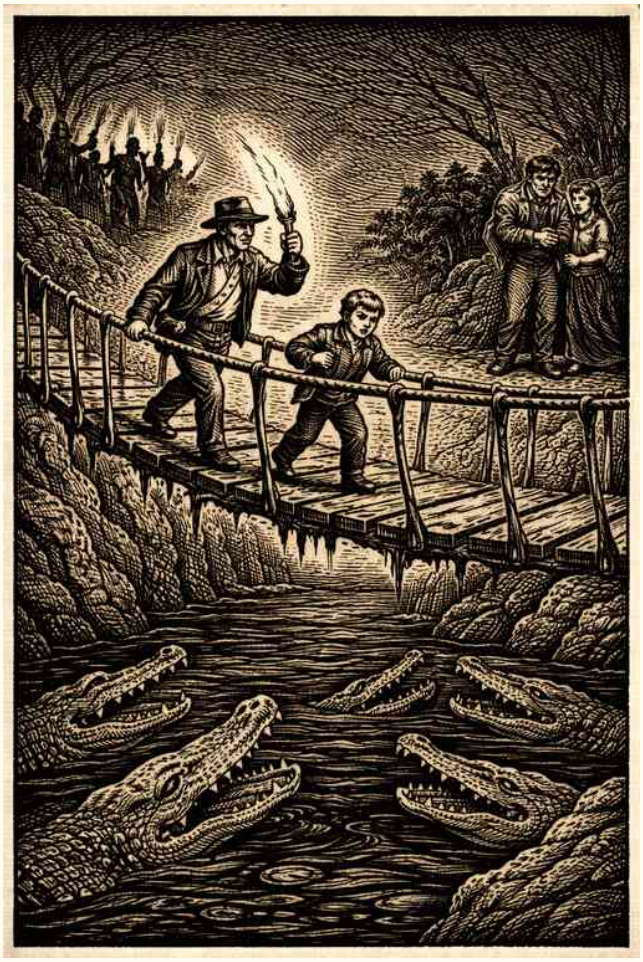
When you have a fully functioning specification that solves the problem, and all the notions are acquired, we'll move on to more complex problems.

Note: I gave you literally all the code, just not in a single copy-and-paste–able block. It's up to you to put it back together.

Note 2: You will *not*, at any point in this course, use use any AI tool to generate or explain TLA$^+$ specifications. That would be utterly pointless.

7 Indiana Jones and the Temple of Verification

Here is another funny problem:



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?

7.1 **EXTENDS**: using / importing other modules . . . 38
7.2 Functions and records . . . 39
7.3 Debugging with `Print` . . . 39
7.4 Tightening the problem's vocabulary . . . 40
7.5 The `CHOOSE` (∋) "quantifier", weird and dangerous? . . . 40
7.5.1 Beware of impossible choices! . . . 40
7.5.2 ∋ vs. ∃: understanding non-determinism . . . 41
7.5.3 Aside on notation, history, and hidden power . . . 42
7.6 No overloading / polymorphism . . . 43
7.7 Identifying inputs / parameters of the problem . . . 43
7.8 **ASSUME**: check your inputs . . . 44

First, what are our actors? 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 `Next`, however. You'll have to figure it out. All in all, we have the actors:

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

Each of them has a certain "time-to-cross" associated to them. We need a convenient structure to store that. Furthermore, we'll want to get the time for the slowest of a group of actors.

In WGC, we dealt with strings and sets of strings. This time, we need to play with integers, and probably something like a Python dictionary, along with computations such as the max function.

Mathematically, we need something like that:

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

Note: *I include* $L \mapsto 0$ *just so we don't need to explicitly exclude* L *from the max computation. If you dislike assigning a* τ *value for* L*, you are welcome to alter the definitions accordingly in your specification.*

How do we do that in TLA$^+$?

7.1 `EXTENDS`: using / importing other modules

First, we are going to need to deal with integers. In Python, you **`import`** modules. In TLA$^+$ you generally use **`EXTENDS`** to much the same purpose.

The modules `Naturals`, `Integers`, and `Reals`, deal with $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$, respectively. Here we'll use `Naturals`, which defines useful things such as arithmetic operators, the set `Nat` (= $\mathbb{N}$), and the notation `a..b` for the integral interval $[\![a, b]\!] = [a, b] \cap \mathbb{Z}$. However, `Naturals` does not define things like the unary minus operator, because it makes no sense for $\mathbb{N}$. It is defined in `Integers`, for $\mathbb{Z}$.

We'll also extend the `TLC` module because it contains, among other things, a `Print` operator for debugging purposes. More on that shortly.

7.2 Functions and records

Now, we need some kind of Python-dictionary–like structure for τ. There is an ideal type for that in TLA$^+$, which is *records*. We're going to write

```
τ ≜ [ I ↦ 1, G ↦ 2, W ↦ 4, K ↦ 8, L ↦ 0 ]
```

Note: *Unlike Python, TLA$^+$ does* not *support Unicode identifiers. You will type* `tau`.

That is a record. Its intuitive meaning is clear, but it is important to understand what's going on more precisely, so we know how to use it.

Records are syntactic sugar for / a special case of *functions*, so let's deal with that first.

Functions, as mentioned before are a data type in TLA$^+$, like, say, sets, and are not to be confused with operators, which are a syntactic construct more akin to C macros. Since they are a data type, you can have **VARIABLES** that are functions, for instance.

In mathematics, functions are traditionally defined as sets of pairs (tuples). Sets and pairs are therefore more primitive constructs, and functions are just syntactic sugar around them. In TLA$^+$, functions are primitives, and things like tuples and records are defined as special cases of functions.

Let us see two function-related primitive TLA$^+$ notations which we'll need soon. In TLA$^+$, if `f` is a function and `e` an expression, then `f[e]` denotes function application, that is to say, it evaluates to the value of the function at the value of `e`. **DOMAIN** `f` evaluates to... the domain of `f`, unsurprisingly. It is a set.

Records are simply functions whose domain is a finite set of strings, with a convenient syntax for definition and access. The "keys" in the record definition serve as string, without needing to be quoted. So our τ is the function which to `"I"` associates the value `1`, i.e. `τ["I"] = 1`, and so on for the other values.

Furthermore, we can just take `A ≜ DOMAIN tau` and have our set of actor strings.

7.3 Debugging with `Print`

Let's take this opportunity to talk about `Print(,)`. It is defined in the `TLC` module as

```
Print(out, val) ≜ val
```

with a note that it causes TLC to print the values *out* and *val*. That raises questions. Where? In *Model-checking results/User output*, whenever you run the model — even if you don't actually check any property. Why? Because you can use it around any part of any expression to check its value. It's a debugging tool. Let's use it to see the value of **DOMAIN** `tau`:

```
A ≜ Print("Hello!", DOMAIN τ)
```

In the output, you will find, as expected

```
"Hello!"  {"L", "I", "G", "W", "K"}
```

Note that `A` will still be correctly defined, because `Print` has the same value as its second argument, by definition. The first argument is there in case you have several types of prints, to have something to differentiate them in the output.

7.4 Tightening the problem's vocabulary

We now have a set of actors. Unlike WGC, we don't have a one-character constant (meaning nullary operator, not to be confused with `CONSTANTS`) for each of them, such as `W` $\triangleq$ `"Wolf"`. Is that a problem? The Lamp has a special role to play, so we should probably have `L` $\triangleq$ `"L"` somewhere — `"L"` is not long to type, but it's good to have some safety in case of typos: the parser will probably tell you that `l` is undefined, but won't see anything wrong with `"l"` or `"L "`. The other actors will probably be picked from `A` via ∃, and dealt with abstractly with τ, so we'll never need to name them directly in the specification.

7.5 The CHOOSE (∋) "quantifier", weird and dangerous?

Speaking of τ, we still need to extend it to sets: $\tau(s) = \max_{a \in s} \tau(a)$. Let's begin with the classical max operator. It's not in any standard module, so I'll just give you the code you need:

```
max(S) ≜ ∋ x ∈ S: ∀ y ∈ S: y ≤ x
```

It's pretty self-explanatory given that, in the code, ∋ is written `CHOOSE`: the max is any element that is greater than all others, which we choose and return.

It's important to understand what's going on, here, not so much for the sake of ∋ itself, but because studying how it differs from ∃ will clarify what we're doing when we're not using it.

`∋ x ∈ S: P`, is (an expression that denotes) a value that's in `S` and satisfies `P`. It can also be written as `∋ x: (x ∈ S) ∧ P` — like for ∀ and ∃, those are called is *bounded* and *unbounded* forms. Both forms are valid TLA$^+$, but TLC can only deal with bounded forms.

7.5.1 Beware of impossible choices!

The behaviour of ∋ is *undefined* if there is no suitable `x`!

In mathematics it is supposed to return a completely arbitrary value; perhaps `"Diplodocus"` or $\frac{\pi}{2}$, who knows...

Thankfully TLC will instead consider that an error and crash with "*Attempted to compute the value of an expression of form* `Ↄ x ∈ S: P`*, but no element of S satisfied P.*", but I recall having had much less explicit error messages in some circumstances.

Whenever you invoke Ↄ, make sure to enforce the existence of a suitable value. That is straightforward enough.

7.5.2 Ↄ vs. ∃: understanding non-determinism

Another, much more subtle, caveat is that the choice is *not* nondeterministic; do not confuse the uses of Ↄ with ∃. Whenever you want to translate things like "the Farmer *chooses* someone to cross with", you (almost always) actually want ∃ and not Ↄ.

To illustrate that, let's go back to WGC and try:

```
Next ≜
    ∃ a ∈ left: left' = left \ {a}
```

That is to say, a valid step is that which removes from `left` *any* actor. If you deactivate all invariants, you get 16 states; from the initial state, four possible transitions, as any of our actors may leave. And so on. You have modelled the situation where all actors have their own boat and can go right.

Contrast that to:

```
Next ≜
    left' = left \ { Ↄ a ∈ left : ⊤ }
```

That is to say, a valid step is that which removes from `left` *an arbitrary actor*.

If you deactivate all invariants, you get a total of five states, in a chain:

```
left ↦ {"Wolf", "Goat", "Cabbage", "Farmer"}
left ↦ {"Goat", "Cabbage", "Farmer"}
left ↦ {"Cabbage", "Farmer"}
left ↦ {"Farmer"}
left ↦ ∅
```

And of course Ↄ crashes on the last one.

What you need to understand is that `Ↄ a ∈ left : ⊤` is a *value*. It's an actor, specifically. There is no such thing as a "non-deterministic value" in mathematics. Given a value of x, the expression $f(x)$ has a value, and only one value; for instance $f(2) = 2$. If you want $f(x)$ to encode something non-deterministic, like "sometimes x, sometimes $x + 1$", then you define $f(x) = \{ x, x + 1 \}$. That's how relations work; $f(x)$ is still one specific value, that happens to be a set, and will never change.

Likewise, `Ↄ a ∈ left : ⊤` is a specific value, that happens to be an actor, a *specific* actor, and will never change. It's just not specified which one it is.

The only two guarantees offered by ↄ are

$$\exists x : P(x) \iff P(\text{ↄ}x : P(x))$$

and that if you choose two values x and y according to two equivalent properties, you get the same value:

$$\big(\forall x : P(x) = Q(x)\big) \implies \big(\text{ↄ}x : P(x)\big) = \big(\text{ↄ}x : Q(x)\big)\ .$$

In particular, if there is no suitable x then $(\text{ↄ}x : P(x)) = (\text{ↄ}x : \bot)$, a completely arbitrary but fixed value.

Getting back to the ↄ-definition of `Next` above, We can guess by the results above how TLC chooses the value in our case (it uses the order in which we defined the set `A` $\triangleq$ `{W,G,C,F}`) but that's, again, not specified, so we can't rely on it, and it really doesn't matter.

∃ works to specify a non-deterministic system because our `Next` encodes the relation

$$\{\,(A, A \setminus \{W\}),\ (A, A \setminus \{G\}),\ (A, A \setminus \{C\}),\ (A, A \setminus \{F\}), (A \setminus \{W\},\ A \setminus \{W, G\}), \ldots\,\}$$

These are exactly the steps / pairs of states that satisfy the formula `Next` (∃-version). That doesn't mean that this version of `Next`, or the existential quantifier, are non-deterministic. `Next` has exactly one value, whether you see it as a Boolean value given an assignment of `left` (in which case it is and will always be a specific value among true or false) or as a function of `left`, in which case it associates to each possible value of `left` a specific set of successors, or as a relation, in which case it is the set of pairs above.

Non-determinism is a property of the state systems we describe using mathematics, not of the mathematics we employ to describe them. Saying that the set $\{x, x + 1\}$ is non-deterministic is complete nonsense. In the context of describing the successors to a state system, however, it may *represent* a non-deterministic choice between two possible successor states.

Bottom line: ↄ is *occasionally* necessary for some helper functions like `max`, but requires extremely careful consideration each time it is used. Don't be too friendly with it. 95% of the time, you want an ∃ instead. When in doubt, mentally replace `CHOOSE` by `FIX`, or `ARBITRARY`. "Let's *fix* an *arbitrary* value that satisfies `P`".

7.5.3 Aside on notation, history, and hidden power

Note that ↄ is my *personal* notation for `CHOOSE`, following the convention of reversed letters for quantifiers: ∀, ∃.

There already exists since 1923 a notation in mathematics called the **Hilbert epsilon**, $\epsilon x\, P$, but it is *very* ugly, not widely known, and does not fit the syntactic structure of TLA$^+$.

While it is debatable whether that construct should be called a quantifier — it is arguably more an operator — it works the same way and is at least as powerful as the traditional ∀, ∃, which can be redefined from it:

$$\exists x : P(x) \iff P(\text{ↄ}x : P(x))$$

Proof. Let $v = \text{Ɔ}x : P(x)$. Either there are some values that satisfy P, in which case v is one of them, and $P(v)$ is true, or there is none, in which case v is some other, arbitrary value in the space of all possible values that might exist, like, say, the string `"Stop! Hammer Time"`, why not, in which case v does not satisfy P: $P(v)$ is false. □

You actually use Ɔ all the time in mathematics, it's just couched in natural language instead of having a notation. Say that you have an automaton with states Q, and want to add a sink state. You don't really care what that sink state is *called*, you just want it to be new, that is, not already in Q. You write "let s be a fresh state" to mean "Let $s = \text{Ɔ}q : q \notin Q$".

Of course TLC does not support the full power of Ɔ; it won't know what to do with the unbounded `Ɔ x: P`, only the bounded form `Ɔ x∈ S: P`, if `S` is finite.

7.6 No overloading / polymorphism

Back on track — we were defining τ on sets — we now have the `max` operator; let

```
τ̄(S) ≜ max( {τ[a] : a ∈ S} )
```

and we're done. You'll write `maxtau` for this function. We can't reuse the name `tau` or "extend" `tau`.

It is very common in maths to use the same symbol τ for a function of domain A and, say, its extension on $\wp(A)$, and the meaning is clearly defined by the type of the function's argument.

This is also a core mechanic of programming languages: method overloading (Python, C++, Java,...), and ad-hoc polymorphism (Haskell, SML,...).

However, that doesn't exist in TLA$^+$, so those are two different functions (actually technically one is an operator!) and we'll need different names (`tau` and `maxtau`), which I render here with slightly different symbols (τ and $\bar{\tau}$).

7.7 Identifying inputs / parameters of the problem

The last line I'll give you is a definition for the maximum time to cross:

```
maxt ≜ 15
```

Why bother when it's faster to write `15`?

First, because I generally hate seeing literals littered all over the code / specification. You'll need to type it several times; what if you write `115` someplace? Give a name to any literal you use more than once, and use that name.

Second, because it's a value we'll probably want to play with. The time we have to cross is clearly a parameter of the question, in that we'll probably want at some point to ask the

same question with, say, 17 minutes, and perhaps with slightly different actors. So we want those things which we may want to change conveniently defined at the beginning, so we can easily tweak them. Generally speaking, if there are literals everywhere, what do you think happens when you mass replace 15 by 17, but some of those instances of 15 represent some other unrelated quantity, that happened to have equal value?

This is actually what **CONSTANTS** / *rigid variables* are for, but we won't use them because **(1)** ergonomically, they are a bit of a pain to use with the toolbox, and **(2)** they are of limited types, and in particular cannot be functions; that excludes τ. This is why we won't use **CONSTANTS** for that purpose in this class, but that doesn't mean you don't need to think about what the natural parameters / inputs of the problem are and define them in such a way that they are easy to tweak. Here, the parameters are `maxt` and τ. `A` is redundant with and derived from τ, and `L` is just an actor that is expected to be present.

7.8 ASSUME: check your inputs

Once you have identified the inputs, derived inputs and the properties they are expected to satisfy in order to make sense, you should check that they really make sense. Think of it as a typing invariant for your inputs, or an assertion in defensive programming (you remember what I told you about **assert** in Python, right?)

For that purpose, the rough equivalent of **assert** in TLA$^+$ is **ASSUME**. What should we assume? Well, that all the times given by either `maxt` or τ should be natural integers: no negative times, and remember that $\mathbb{R}$ is infinite, that floating-point computation is tricky, and that our model must boil down to a *finite* and *reasonably sized* automaton. In model-checking, even if we deal with continuous phenomena, we abstract them away to discrete models.

Also, it doesn't hurt to verify that `A` is indeed the set of actors, as defined implicitly by the domain of τ. This can be done very simply by $\tau \in [A \to \mathbb{N}]$, meaning "$\tau$ is in the set of all (total) functions from A to $\mathbb{N}$". We check both inputs and outputs that way; we could do all that with $\forall$, of course, but this is shorter and cleaner.

Lastly, we should probably ensure that the Lamp is defined and accounted for, since it has a special role to play. An instance of the problem without the Kid would make sense; an instance with the Lamp... not so much.

Thus we write:

```
ASSUME
    ∧ τ ∈ [A → ℕ]
    ∧ maxt ∈ ℕ
    ∧ L ∈ A
```

8 LAB CLASS: Indy

So, to recap the previous section, below is all the code I gave you, in the right order and structured correctly inside a template.

DO NOT COPY AND PASTE FROM THE PDF: this causes all kinds of encoding problems. Copy the source from the text file `Verif4A.code.tla`, available on Celene.

In all similar exercises, your specification will follow the same structure and naming conventions (`Type`, `Init`, `Next`, `Inv`, `Target`).

I have provided fake variables (no, sorry, the variables for this problem are not actually `foo` and `bar`, it's up to you to find them) to illustrate the kind of documentation I'd expect. The variables are the centrepiece of your specification; once you understand what your states *are*, everything else should be more or less straightforward.

Note the `Type` invariant. It's one of those things that are not part of TLA$^+$'s syntax, but which are pretty customary and which, for the sake of consistency, I shall impose in this course.

In the previous exercise, we mixed our typing invariant `left ⊆ A` with a more high-level invariant `l(left)` in the same operator `Verif`. We did it that way because introducing the notion of invariant before `Init` or `Next` would not have gone well, but now that we know what an invariant *is*, we see that the right place to put the typing invariant, which we shall always name `Type`, is, of course, right after the definition of the variables.

Let's clearly distinguish the roles of the invariants:

`Type` just types the variables. Maybe sets reasonable bounds for them, as I did in `bar ∈ 0..50`, and tests basic consistency, e.g. if in WGC I had two sets `left` and `right`, I might want to check `left ∩ right ≠ ∅`. `Type` protects mostly against trivial errors at the type of writing the specification.

At the very end of the file, `Inv`, along with `Target` (and perhaps other problem-specific properties), deal with anything else. For WGC, the action of `Verif` (specifically the non-typing part ofg it, `l(left)`) and `Target` combined boils down to "the system can reach a certain state without violating certain constraints". This is a much more complex property than what `Type` deals with; it pertains to the *global behaviour* of the system. `Inv` etc protect against deeper errors in the *design of the system* which the specification describes.

Examples of properties that clearly belong in the last section, not to the typing invariant:

I gave constraints to the Farmer; can he actually go home?

I gave a torch and 15min to Indy; but is he actually doomed?

I designed a radiotherapy firmware full of mutexes that avoids race conditions (e.g. two

processes accessing the same memory at the same time); but does it freeze the machine? And does it actually even avoid race conditions?... If it does not do any of those bad things... does it actually... *do*... anything? (a protocol that does nothing is pretty bug-free).

Now, here is the code; have at it!

```
---------------------- MODULE Indy ----------------------------------------
EXTENDS Naturals, TLC

(***************************************************************************)
(* PROBLEM PARAMETERS / INPUTS                                             *)
\* τ ∈ [Actors → ℕ]:
\*     defines actors set and time to cross for each actor
\* maxt ∈ ℕ:
\*     total time available to cross
(***************************************************************************)

τ ≜ [ I ↦ 1, G ↦ 2, W ↦ 4, K ↦ 8, L ↦ 0 ]
maxt ≜ 15

(***************************************************************************)
(* DERIVED INPUTS,  HELPER FUNCTIONS, AND INPUT TYPING                     *)
(***************************************************************************)

A ≜ DOMAIN τ
L ≜ "L"

max(S) ≜ ℑ x ∈ S: ∀ y ∈ S: y ⩽ x
τ̄(S) ≜ max( {τ[a] : a ∈ S} )

ASSUME
    ∧ τ ∈ [A → ℕ]
    ∧ maxt ∈ ℕ
    ∧ L ∈ A

(***************************************************************************)
(* STATE VARIABLES                                                         *)
\* foo ∈ ℘ Fool: set of fools that are fuzzy
\* bar ∈ ℕ: number of customers at the bar
(***************************************************************************)

VARIABLES foo, bar
Type ≜
    ∧ foo ⊆ Fool
    ∧ bar ∈ 0..50

(***************************************************************************)
(* INITIAL STATE                                                           *)
(***************************************************************************)

Init ≜
```

```
    something that makes sense

(*****************************************************************************)
(* TRANSITIONS                                                               *)
(*****************************************************************************)

Next ≜
    who knows?

(*****************************************************************************)
(* PROPERTIES: INVARIANTS & REACHABILITY TARGETS                             *)
(*****************************************************************************)

Inv ≜
    your invariant here for $0.99 a month

Target ≜
    stuff you want to reach, like, the Moon or something, idk
=============================================================================
```

Tip: you should expect 185 states, and a length 6 success trace, with 0 time remaining.

This is where visualising the state-machine graph shows its limits.

In previous versions of this course (pre 2025) I used my own tools, based on my NFA library [(h)], and we could trim the state space.

That is to say, we could remove all the states that are not on the way of a solution, all "dead-ends". That reduced the state space from 185 to 8, and it was easier to understand what's going on, and infer the winning strategy for that problem.

It is possible to export the graph from TLC / the Toolbox to `dot` format, so I could use a Python libary like pygraphviz, graphviz-python, pydot, or graphviz, to import that to nfalib, trim and show.

I'm probably not going to do that, and will show you the result instead, because you would need to install and manipulate two pieces of software instead of one, and that takes valuable class time.

Just something to think about when your graph is large, but not so huge that you can't generate it at all: you can still often extract relevant information from it, if you have the tools.

[(h)] https://github.com/vincent-hugot/nfalib

9 Semaphores, Peterson's Algorithm, and Temporal Logics

9.1 Semaphores 49
9.2 Semaphores as a PlusCal algorithmic specification 51
9.3 PlusCal in the toolbox: understanding the translation 52
9.3.1 Program Control 54
9.3.2 The `Spec` formula was here all along 54
9.4 Invariants were Temporal (Safety) properties all along! 55
9.5 Liveness Properties: poor starving semaphores! 57
9.6 You can't avoid thinking about the underlying system 58
9.7 Peterson's algorithm 59
9.8 Safety: no problem 60
9.9 What you want, you get! 61
9.10 . . . but who cares what you want? 62
9.11 Fat or starving, at the Scheduler's mercy 63
9.12 The Temporal Logics: CTL*, CTL, & LTL 63

Let's mark a pause in our funny river-crossing problems and move on to more computery stuff, for a while.

At this point, we begin to see how to model *systems*; now we focus on how to express the *properties* we need to verify about them, beyond just invariants.

The aim of this section is to motivate the need for **temporal logics**, even if **we won't understand all the details**. We'll also get a taste of how to model concurrent processes in practice, even if, again, we won't understand all the details and I won't ask you to master that.

Nor will I ask you to study the temporal aspects of TLA$^+$ (the T in TLA); we'll instead move on to Sec. 28$_{[p170]}$: "Classical and Temporal Logics: (W)S1S, CTL*, CTL, LTL", and focus on Sec. 28.2$_{[p183]}$: "CTL*: Computation Tree and Linear Time Logic (CTL+LTL+$\cdots$)" and, specifically for the exam, Sec. 28.2.5$_{[p189]}$: "CTL: Computation Tree Logic".

TLA$^+$ is based on those logics (specifically, LTL, which is a fragment of CTL*) but with additional complexities[(i)] which we won't have time to deal with. Studying CTL*/CTL instead will let us understand 90% of the temporal aspects of TLA$^+$, and that will be enough for our purposes.

9.1 Semaphores

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:

[(i)] Chiefly stutter-invariance.

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 (no queue) 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
```

9.2 Semaphores as a PlusCal algorithmic specification

Let's model this in TLA$^+$. We could do this directly, but instead we'll use an algorithm language called PlusCal.

> *PlusCal is an algorithmic language that has the "look and feel" of imperative pseudo-code for describing concurrent algorithms. It has a formal semantics, through translation to TLA+, and algorithms can be verified using the TLA+ tools.*
>
> — *PlusCal / TLA+: An Annotated Cheat Sheet*, Stephan Merz.

I will not require you to write or understand the semantics of PlusCal in any detail. We are firmly waving our hands in this section.

Here is an "implementation" of the above pseudo-code in PlusCal:

```
N ≜ 2
P ≜ 1..N

--algorithm Semaphore
{
    variables
      sem = 1;

    fair process (p ∈ P)
    {
      NC:   while (⊤)
            {
                skip; \* NonCritical Section
                await sem > 0;
                sem := sem - 1;
      CS:       skip; \* Critical Section
                sem := sem + 1;
            }
    }
 }
```

It is fairly straight-forward to understand.

(1) You can ignore the **fair** keyword; *in this context*, it just boils down to "the system won't get stuck in the same state forever". In other words, it elides a difficulty which you didn't know we had (stutter-invariance) because I hid it from you so far... and I shall continue to hide it, mostly.

For the purpose of this course, accept that there are rather mysterious things called "stutters" and "fairness properties", which are subtle and difficult to master even if you already know temporal logics, and that I will occasionally use some obscure keyword or formula to *make things behave the way you intuitively expect them to*.

(2) `await` just blocks the execution until the condition becomes true.

(3) The labels `NC:` and `CS:` basically serve as states for the underlying automata.

I don't have to put labels everywhere. The idea is to put them wherever the process can be interrupted / preempted.

For instance, using a label `DEC: sem := sem - 1;` would be wrong, here, as we specifically want the `P(sem)` operation to be atomic.

The choice of labels is therefore quite significant, as it determines the atomicity of our algorithm.

(4) `skip` does nothing at all. I could just have put a comment here, but `skip` gives an opportunity to put a label there if one wants.

9.3 PlusCal in the toolbox: understanding the translation

To use PlusCal in practice, you write the algorithm in a comment, and the Toolbox then generates the corresponding TLA$^+$ code in the same file on demand (CTRL+T by default), between comments

```
\* BEGIN TRANSLATION
...
\* END TRANSLATION
```

`\* BEGIN TRANSLATION` can be followed by checksums to ensure that the PlusCal and the TLA$^+$ code are synchronised.

You can put definitions before and properties after. Here is my module:

```
------------------------------ MODULE Semaphore ------------------------------
EXTENDS Naturals

N ≜ 2
P ≜ 1..N

(*
--algorithm Semaphore
{
    variables
      sem = 1;

    fair process (p ∈ P)
```

```
    {
      NC:   while (⊤)
            {
                skip; \* NonCritical Section
                await sem > 0;
                sem := sem - 1;
      CS:       skip; \* Critical Section
                sem := sem + 1;
            }
    }
 }
*)

\* BEGIN TRANSLATION (chksum(pcal) = "21cfb14a" ∧ chksum(tla) = "369d462f")
VARIABLES sem, pc

vars ≜ ⟨ sem, pc ⟩

ProcSet ≜ (P)

Init ≜ (* Global variables *)
        ∧ sem = 1
        ∧ pc = [self ∈ ProcSet ↦ "NC"]

NC(self) ≜  ∧ pc[self] = "NC"
            ∧ ⊤
            ∧ sem > 0
            ∧ sem' = sem - 1
            ∧ pc' = [pc ‖ ![self] = "CS"]

CS(self) ≜  ∧ pc[self] = "CS"
            ∧ ⊤
            ∧ sem' = sem + 1
            ∧ pc' = [pc ‖ ![self] = "NC"]

p(self) ≜ NC(self) ∨ CS(self)

Next ≜ (∃ self ∈ P: p(self))

Spec ≜  ∧ Init ∧ □[Next]_vars
        ∧ ∀ self ∈ P : WF_vars(p(self))

\* END TRANSLATION

Type ≜
    sem ∈ 0..1

C(i) ≜ pc[i]="CS"

Race ≜ ∃ i,j ∈ P: i ≠ j ∧ C(i) ∧ C(j)
```

```
Inv ≜
    ¬Race

Safety ≜
    □¬Race

Liveness ≜
    ∀ i ∈ P : ◇ C(i)
================================================================================
```

9.3.1 Program Control

We currently lack a few elements of syntax to fully understand the translation. We will understand everything (except `Spec`) after Sec. 11[p75]: "LAB CLASS: Three_Islands".

In the meantime, intuitively, `pc` means "program control", and is a function

```
pc ∈ [1..2 → {"NC", "CS"}]
```

which stores the current states of both processes. A line like

```
pc' = [pc ‖ ![self] = "CS"]
```

means "the state of the process `self` becomes `"CS"`; the other process does not change".

So, in nutshell

```
NC(self) ≜  ∧ pc[self] = "NC"
            ∧ sem > 0
            ∧ sem' = sem - 1
            ∧ pc' = [pc ‖ ![self] = "CS"]
```

means that, if you're a process, and you are currently in your NonCritical section (`pc[self] = "NC"`), and the resource is available (`sem > 0`), you can decrement `sem` and move on to your critical section. (The additional $\top$ comes from the **`skip`**.)

9.3.2 The `Spec` formula was here all along

As for `Spec`,

```
Spec ≜  ∧ Init ∧ □[Next]_vars
        ∧ ∀ self ∈ P : WF_vars(p(self))
```

we will not fully understand it in this course, but $\Box$ is a temporal logic modality read "always", which we will deal with and understand in the context of CTL*. We won't focus on the `[Next]_vars` syntax, though; that is TLA$^+$ specific and related to *stutters*, which I don't want to deal with here.

The second part, `∀ self ∈ P : WF_vars(p(self))`, is here because of the mysterious **`fair`** keyword I used. For our purposes it makes those pesky stutters go away, whatever they are. Let's all close our eyes and pretend it is not here, OK.

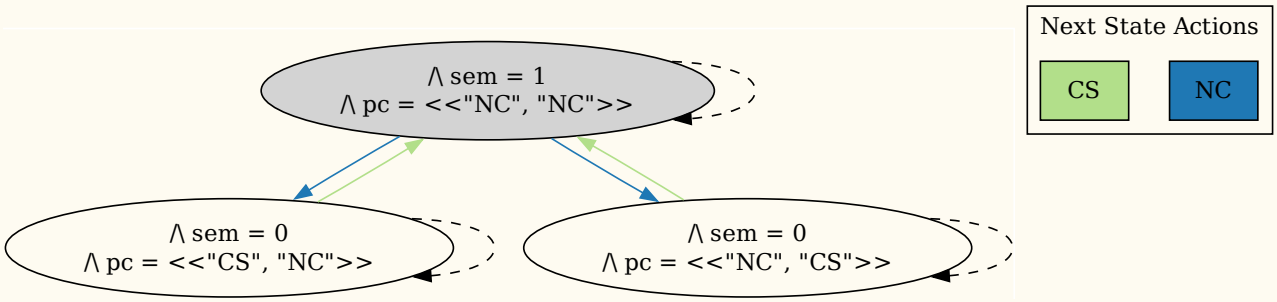
Intuitively, `Spec` is the temporal formula describing all valid traces of the system described by our specification. Indeed, it *is* our specification. The toolbox will be configured to use it instead of `Init` and `Next` — so that the mysterious fairness condition can also be taken into account.

Simplifying `Spec` into `Init ∧ □Next` for the sake of discussion, it is read "`Init` and always `Next`". It is true of traces (that is to say, infinite sequences of states) such that the very first state satisfies `Init`, and every transition (that is, couple of successive states) satisfies `Next`.

In other words, `Spec` describes a set of traces; the automaton you see when you "run" a Model is a representation of the set of traces described by `Spec`. They are really just different ways of defining/describing/visualising the same thing, which is the abstraction of our system.

Whenever we feed `Init` and `Next` to the toolbox, it's just a shortcut for writing `Spec` as above.

In the case of the current system, the automaton visualising `Spec` is



You can ignore the little self-loops on every states. Those are the "stutters" I mentioned. They didn't appear in the automata for WGC and Indy because we didn't deal with properties for which they might matter. Here, they *might* matter, but I made them *not* matter by the magic of the **`fair`** keyword, so we'll just pretend they don't exist.

9.4 Invariants were Temporal (Safety) properties all along!

We see on the graph that race conditions are, fortunately, avoided, thanks to the semaphore. That is to say, there is no state where both processes are in their critical sections simultaneously.

This is checked by our invariant `Inv`, which holds.

```
C(i) ≜ pc[i]="CS"
```

```
Race ≜ ∃ i,j ∈ P: i ≠ j ∧ C(i) ∧ C(j)

Inv ≜
    ¬Race
```

For `N ≜ 2`, I could just have written

```
Race ≜ C(1) ∧ C(2)
```

It just means "there is a race condition / critical conflict".

Then we have our first temporal property:

```
Safety ≜
    □¬Race
```

You can put it in the Model in the *Properties* box in the toolbox, below *Invariants*. Run the model, you will find that it holds.

In fact, it does exactly the same thing as `Inv`. TLA$^+$ temporal properties are predicates on paths/executions/traces/behaviours of the system, and implicitly quantified universally, which is to say they must hold for *all* traces.

This particular property, `□¬Race`, is read "Always, not `Race`", or "Never `Race`". That is to say, every state in the trace must satisfy `¬Race`, which is to say that none may satisfy `Race`. In other words, at no point in the behaviour are you allowed to have a race condition.

So defining an invariant `I` in the toolbox is really a shortcut for testing against the temporal property `□I`.

Taking a step back, everything we do, defining `Init`, `Next`, and invariants `I1`, `I2`, and pressing the button to model-check that, is really window-dressing for testing whether the following formula holds:

```
Init ∧ □[Next]_vars ⇒ □I1 ∧ □I2 ∧ ...
```

In other words, for all possible behaviours, if it is a behaviour of our system (i.e. defined by `Init` and `Next`), then it must satisfy our invariants.

Invariants are by far the most practically relevant among temporal formulæ, and also the easiest to grasp intuitively.

They belong to the category of **safety properties**, which boil down to "something bad never happens". "The plane does not crash". "The intruder does not get root access to the system" are examples of safety properties (and invariants). [j] Such properties are very very important, but not quite enough on their own.

[j] There are formal definitions for safety and liveness properties, but we don't need to go there.

9.5 Liveness Properties: poor starving semaphores!

It's very easy to achieve perfect safety: just do nothing. Ground the plane permanently: it will never crash. Turn off the computer: the intruder will never have root access.

This is a perfect solution for tech support and IT departments everywhere: make the system so inconvenient to use that nobody uses it. No users = no problem = easy life.

The rest of us also want our systems to *actually do something at some point*, and for this we need **liveness** properties. Those boil down to "something good eventually happens". For instance, "The plane *will* land at my destination", or "If I need root access, I'll get it, eventually".

Usually what we really want is for something good to happen at some point, *without* something bad happening in the meanwhile. We need both safety and liveness.

Sometimes, we only need the most trivial possible property: the reachability of some state(s). Can we reach `left = ∅`? It's not even really a liveness property; it tells us that it's possible for something good to happen, if the right steps are taken; liveness tells us something good is guaranteed to happen. As we have seen, reachability is the dual question to invariance.

Sometimes, however, we really need to deal with more complex liveness properties. Let's go back to semaphors.

They are safe, we have established that. That does not mean there is no possible problem, however. Nothing prevents the resource from being granted to the same process again and again, while the other starves until the end of time.

This is the notion of **starvation**, as opposed to **race conditions**; two very different problems, yet equally bad to run into.

This is formalised by the following property:

```
Liveness ≜
    ∀ i ∈ P : ◇ C(i)
```

It states: for all behaviours of the system (that is implicit), for each processes `i`, *eventually*, at some point in the execution, `C(i)` holds.

That is to say, each process is guaranteed to enter its critical section at least once, no matter what happens.

This property is *not* satisfied. The toolbox will give you the following counter-example:

```
⟨
[
 pc ↦ ⟨"NC", "NC"⟩,
 sem ↦ 1
],
```

```
[
 pc ↦ ⟨"CS", "NC"⟩,
 sem ↦ 0
],
Back to the first state...
⟩
```

Note that that describes an infinite trace, looping between two states. It is a valid trace of the system, such as we have described it, and it is such that the second process never gets to play.

Such a scenario can happen if, say, your OS scheduler decides to completely ignore a process, for some reason. Of course schedulers are designed to avoid that inasmuch as possible.

9.6 You can't avoid thinking about the underlying system

Fairness properties are essentially liveness properties that deal with the assumptions that can be made about the system / scheduler on which we run.

We cannot completely avoid making assumptions: if the system on which we run is frozen and stutters infinitely on the initial state, it hardly matters how clever our algorithm for "guaranteeing liveness" is. We have to *at least* assume we run on a computer that . . . *actually runs*.

This is what I sneakily did with magical **`fair`** keyword. Without it, the counter-example would just have been

```
⟨
[
 pc ↦ ⟨"NC", "NC"⟩,
 sem ↦ 1
]
,
Stuttering...
⟩
```

That is to say, we're stuck in the initial state forever and ever (it's the infinite trace looping on the initial state).

That's not something we have explicitly taken into account in our `Next`, but TLA$^+$ is forced to care about it anyway, so I made it go away with **`fair`**.

Actually, in general **`fair`** means something much stronger than "the system is not frozen". It just boils down to that *in this specific example* (and won't in the example in the next section, so I'll write down a more specific property for that).

Fairness properties in general are *subtle and complicated*, and I won't go into details, or ever ask you to write them, but I'll just make sure in our examples that we deal with a system

that's not utterly frozen, i.e. avoiding infinite stutters.

9.7 Peterson's algorithm

Peterson's algorithm, like semaphores, avoids race conditions, and additionally guarantees starvation-freeness, and even bounded waiting, under certain conditions.

Let's take a look at (one version of) it.

```
P ≜ 1..2 \* set of 2 processes
other(x) ≜ 3-x \* get other process

--algorithm Peterson
{
  variables
    want = ⟨⊥,⊥⟩,
    turn = 1;

  process (p ∈ P)
  {
    NC:     while (⊤)
            {
                skip;  \* Noncritical Section
                want[self] := ⊤;
    Iwant:      turn := other(self);
    Iwait:      await want[other(self)] = ⊥ ∨ turn = self;
                skip;  \* Critical Section
    CS:         want[self] := ⊥;
            }
  } \* end process
} \* end Peterson
```

The basic idea is that of instead just a mutex, you have a `turn` variable saying whose turn it is, *and* a `want` variable for each process used for signalling interest in accessing the resource. `want` is a tuple, associating a Boolean to each process.

Note that the test

```
await want[other(self)] = ⊥ ∨ turn = self;
```

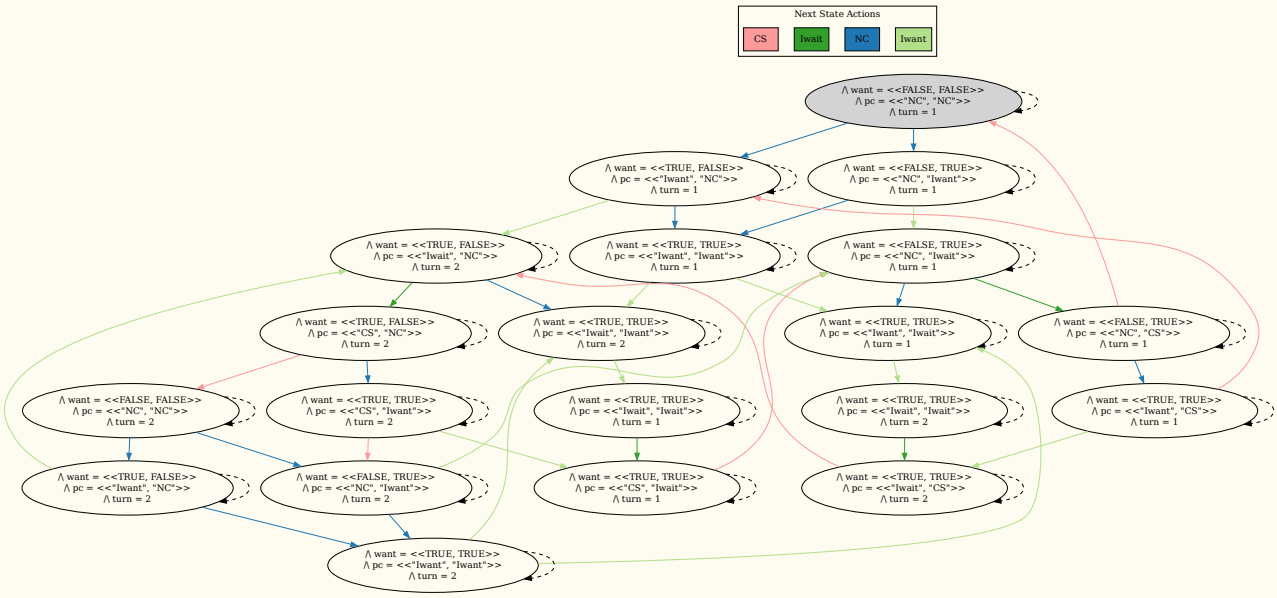
is a atomic in my version, which means we generously assume that the process is not preempted between testing `want` and `turn`. I did this for the sake of simplicity, but the real algorithm can be made to work without that.

Note that, compared to semaphores, I dropped the **`fair`** keyword; this is because in this context it would mean something stronger than "the system does not freeze", which is all I want.

Apart from that, there is nothing new to see here, syntactically. The algorithm is just a bit

more complicated than before, and it's a bit harder to understand how/whether it will work, intuitively.

What does the transitions system look like?



Oh. OK. Twenty states. It's a bit harder to see what's going on. (And it would be so much worse had I not simplified the algorithm a bit.)

9.8 Safety: no problem

Let's see how this bad boy behaves. Let's start with our invariants. Beyond typing, we do exactly the same thing as for Semaphores.

```
Type ≜
    ∧ want ∈ [P → 𝔹]
    ∧ turn ∈ P

C(i) ≜ pc[i]="CS"

Race ≜ ∃ i,j ∈ P: i ≠ j ∧ C(i) ∧ C(j)

Inv ≜
    ¬Race

Safety ≜
    □¬Race
```

And fantastic, it is satisfied. No race conditions, yay! Indeed, if we look carefully at the graph, there is no state of the form

```
pc = ⟨"CS", "CS"⟩
```

but it's better to have the tool confirm it than to rely on our eyes.

9.9 What you want, you get!

Now, let's deal with starvation. As mentioned before, we can't prove any liveness property under no assumption regarding the underlying system / OS scheduler. If the whole computer is frozen, or if both our processes are frozen because the OS focuses on other processes, like, mining cryptocurrency or something, until the end of time, then there is no amount of cleverness *within our algorithm* that can force the scheduler to give us CPU time.

So we are going to assume "the computer is not frozen", which is to say, "we can't stay in the same state continuously forever, if a transition is available, it will eventually be taken".

```
NoFreeze ≜ WF_vars(Next)
```

`WF` stands for *weak fairness*. It's a shortcut for a useful formula of temporal logic, which I'm not going to explain here. Just "trust me, bro" when I say that `NoFreeze` means what it says on the tin.

So, all of our liveness properties will be of the form "`NoFreeze` ⇒something". This was already the case for Semaphores, but in that case I could hide it more sneakily by using the **`fair`** keyword.

Without further ado, let's see our first liveness property:

```
Liveness ≜
    NoFreeze ⇒ ∀ i ∈ P : □( want[i] ⇒ ◇ C(i) )
```

This means: assuming the computer is not frozen, both processes have the following guarantee, no matter what happens: if at any point they ask for access to the resource, eventually they'll get access (enter their critical section).

The pattern

$$\Box(X \implies \Diamond Y)$$

is so common that there is a shortcut notation for it in TLA$^+$: `X ⇝ Y`. I don't know its official name, but I read it as "X leads to Y". It means "Always / At any point, if you have X, then sometime later you'll get Y". Thus our liveness property could equivalently be written

```
Liveness2 ≜
    NoFreeze ⇒ ∀ i ∈ P : want[i] ⇝ C(i)
```

Running the model, we see that this property is indeed satisfied. Victory is achieved.

9.10 . . . but who cares what you want?

Can we get something even stronger? Since any process that asks is guaranteed critical access, and those processes have nothing else going on in their lives but to loop infinitely taking and releasing access, shouldn't that mean that they'll both obtain critical access infinitely?

This concept is written $\Box\Diamond X$ in temporal logic, and called "infinitely often".

"Always, Eventually X". That is, no matter where you are in the execution, it's always true that you have an X somewhere in your future. So there can't be a last time you see X, because at the moment after that, you still have an X in your future, somewhere.

So, our intuition tells us that since `Liveness`, above, is true, it should also be true that each process has critical access infinitely often.

```
Liveness3 ≜
    NoFreeze ⇒ ∀ i ∈ P : □◇ C(i)
```

Looks good. Press the button and. . . huh oh, the toolbox does not agree with our intuition, and gives us an infinite trace where Process 1 never has critical access:

```
⟨
[
 pc ↦ ⟨"NC", "NC"⟩,
 turn ↦ 1,
 want ↦ ⟨⊥, ⊥⟩
],
[
 pc ↦ ⟨"NC", "Iwant"⟩,
 turn ↦ 1,
 want ↦ ⟨⊥, ⊤⟩
],
[
 pc ↦ ⟨"NC", "Iwait"⟩,
 turn ↦ 1,
 want ↦ ⟨⊥, ⊤⟩
],
[
 pc ↦ ⟨"NC", "CS"⟩,
 turn ↦ 1,
 want ↦ ⟨⊥, ⊤⟩
],
Back to state 1
⟩
```

Basically, Process 1 never moves, while Process 2 eats infinitely often. We have shown that *if you ask for access* you are granted access, but here the scheduler completely and utterly ignores Process 1, so that *it never even has an opportunity to ask* for access.

But wait, if nothing prevents the scheduler from being so unfair as to completely ignore a process forever, how did we even manage to enforce `want[i] ⇝ C(i)`?

Well, it's because the scheduler is forced to advance *some* process by our `NoFreeze` assumption. So eventually some process will set `want`. When that's done, eventually the other process will be blocked by **`await`**, and the scheduler will have no choice but to take the only available transition, which is that of the process with `want` set.

9.11 Fat or starving, at the Scheduler's mercy

That makes sense; and that gives us the intuition that a weaker property holds: the scheduler can starve *whichever* process it chooses, but it cannot starve *both*, or indeed even prevent at least one of them from having critical access infinitely often:

```
Liveness4 ≜
    NoFreeze ⇒ ∃ i ∈ P : □◇ C(i)
```

And this property… does indeed hold. Whew!

That looks like a pretty damn good guarantee given how weak our "no freeze" assumption is… but then again it's also true of Semaphores :-)

More interestingly, with the **`fair process`** keyword, the assumption would have been "the scheduler cannot ignore a process continuously and infinitely", which is a pretty safe assumption for any scheduler that is not *designed* to hate your guts, and `Liveness3` would have been satisfied.

Even that assumption did not work nearly as well for Semaphores, though. Even *with* **`fair`**, we failed `∀ i ∈ P : ◇ C(i)`, let alone `∀ i ∈ P : □◇ C(i)`. We could spend hours discussing the finer points of fairness properties, but that's not the goal here.

I hope that you have now some intuitions as to what temporal properties are, and the kind of details you need to pay serious attention to when you're trying to design a system / algorithm that you really don't want to fail.

9.12 The Temporal Logics: CTL*, CTL, & LTL

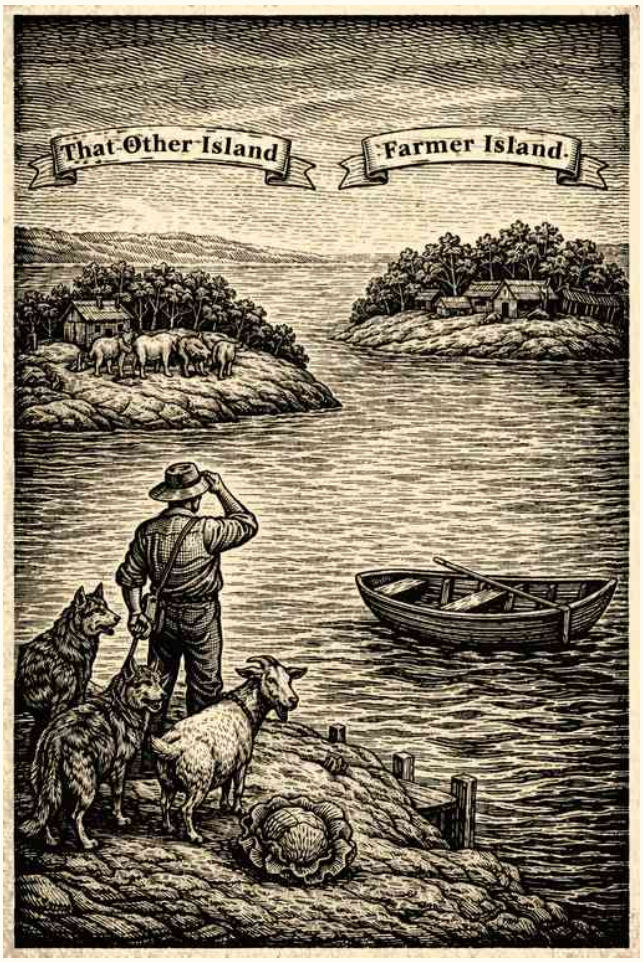
Now we'll focus on Sec. 28[p170]: "Classical and Temporal Logics: (W)S1S, CTL*, CTL, LTL", and specifically Sec. 28.2[p183]: "CTL*: Computation Tree and Linear Time Logic (CTL+LTL+···)" and, for the exam, Sec. 28.2.5[p189]: "CTL: Computation Tree Logic". We'll get a much better understanding of □ and ◇ in the process.

Checking CTL formulæ will account for one half of the final exam.

Question on TLA and general points developed in the lectures and lab classes, which continue in the next section, will account for the other half.

10 The Three Islands, the Two Wolves, the Goat, and the Cabbage

This is your *favourite* problem again, 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?

10.1 Vocabulary and Inputs 67
10.1.1 Abstraction is cutting away irrelevant details: do more for less 67
10.1.2 A good abstraction often has fewer primitive concepts 68
10.2 **VARIABLES**: `[A → Locs]` vs. `[Locs → ℘ A]`, for you and TLC 69
10.3 Fifty shades of `Init`: writing and altering functions 71
10.3.1 TLC doesn't know what you know 71
10.3.2 No convenient ↦ notation outside of records 72
10.3.3 Introduction to the hidden gems ↦ (:>) and ▶(@@) 72
10.3.4 The not-quite λ-expression $[x \in S \mapsto e]$ 73
10.3.5 **IF THEN ELSE** 73
10.3.6 **CASE ... OTHER** 74
10.3.7 The winner: tuples and sequences 74

10.1 Vocabulary and Inputs

We have three locations our actors can be in. While the obvious move is to give them fancy names like `"Magical Market Island"` etc, remember the sayings of the Wise[k]: **(1)** laziness is a virtue, **(2)** measure twice, cut once, and **(3)** in computer science there are only four numbers: 0, 1, 2 (sometimes), and N,

10.1.1 Abstraction is cutting away irrelevant details: do more for less

By **(2)**, let's spend some time thinking before writing anything. Thinking about it, we don't care which island is which, so long as we can distinguish them. By **(1)**, let's name them $1, 2, 3$. Thus we have our set of locations: `Locs = 1..3`.

Is it a problem input? Well, not really, that seems a fixed quantity of the problem; "Three" is the problem's title, after all. But by **(3)**, there is no such thing as the number 3. Indeed, thinking about it, we see the problem: in WGC we were about to encode our states in a single set `left` $\subseteq$ `A` because there were only two bank. Now that there are three, we need either three sets, or I guess two and we deduce the third by complement.

Either way, it's getting ugly, and probably uglier than having a single location function `l` $\in$ `[A` $\to$ `1..3]`. And if we do that, it does not cost us more effort to have `l` $\in$ `[A` $\to$ `1..N]` — and we will have than anyway, because the teacher will yell at us if we leave the literal 3 unnamed — and suddenly we have generalised the problem to an arbitrary number of islands, at no extra cost.

One should certain resist the temptation of over-generalising. Go overboard, and suddenly you have generalised WGC to "What is the meaning of life" and things get weird.

One should *gratefully and enthusiastically* indulge in generalisation when it costs us little or nothing to do so — the cost of learning something new not included, it's an investment.

Being on the lookout for abstraction helps us getting rid of irrelevant details, like the names of the islands. Had we *under*-generalised and just written the obvious

```
Islands = { "Market", "Other", "Home" }
```

and proceeded from there, we'd have spent *more time* solving one very specific problem with very specific code which we'd have spent even more time reworking completely for the inevitable next section: "The Four Islands Problem".

That said, in the course of this section, we'll waffle a little bit about sticking to the generalisation to `N`. We'll do it when convenient, and write things depending on `N=3` when it's shorter. By the end, we'll have all the tools to do it fully.

[k] By which I mean me, last year, in the Python course.

So... what is our input, again?

```
N ≜ 3
```

That's it. From that we derive the set of locations:

```
Locs ≜ 1..N
```

Now, what about the actors? Let's not rush and write `A` explicitly; maybe we'll derive it from something else, as we did from τ in Indy.

10.1.2 A good abstraction often has fewer primitive concepts

What must we know about the actors? Well, their conflicts, for one thing. With two wolves it's starting to get complicated, and the writing is on the wall: if we don't generalise *somehow*, the evil teacher is going to come up with "The Fifteen Wolves, The Bazooka, The Smelly Cheese, and the Otaku". We could have a set of wolves, a set of goats, and a set of cabbages, but... while this is a clear generalisation, is sounds both overly complicated as a solution for *this* problem and somehow still *extremely specific*. What if the other problem's conflicts do not follow exactly those patterns with different names?

Let's keep things simple and write the input

```
Unsafe ≜ { {"W1","G"}, {"W2","G"}, {"G","C"} }
```

We have groups of actors that have conflicts. We write those groups. It's short enough to write *in extenso* for this problem, and if there were 15 Wolves then we could write something like

```
Unsafe ≜ { {W,"G"} : W ∈ Wolves }
```

to capture that compactly.

So that solution is both much simpler than "a set of wolves, a set of goats, and a set of cabbages", and more powerful, as we can handle the same problems for the same cost, and many more besides that do not fit the exact same conflict structure. The encoding of that structure is delegated to the input. A good compromise.

`Unsafe` also contains all the actors except the Farmer. We could just add him, but while we are at it, what if there were other actors not engaged in any conflicts? Since we have an `Unsafe` set, we might as well have a `Safe` set, and leave open the possibility that the Farmer himself may have conflicts. We could generalise to have possibly several Farmers, but that sounds more complicated, so we'll pass on that; since the Farmer has a special role, we should have an identifier for him. It's not really a meaningful input, but it needs to be *somewhere* early, because we need it to be defined for `Safe` and `Unsafe`, so we might as well let it be an input, in case the user wants another name for the Farmer, why not.

So we let:

```
F      ≜ "F"
Safe   ≜ {F}
```

and then we can derive `A`. It contains the Farmer, Safe actors, and Unsafe ones. So we need

$$\bigcup \texttt{Unsafe} \;=\; \bigcup_{S \in \texttt{Unsafe}} S\,.$$

In ASCII TLA$^+$, this is written `UNION Unsafe`. We have:

```
A ≜ Safe ∪ ⋃Unsafe
```

We should probably make sure our inputs make sense. The Zero Island problem and the $\frac{\pi}{2}$ Islands Problem are cute, but not cute enough to consider. If an actor is Safe but also Unsafe, I don't know what those words mean anymore, so let's not have that. The Farmer is kind of important, we should have him somewhere.

```
ASSUME
    ∧ N ∈ ℕ \ {0}
    ∧ Safe ∩ ⋃Unsafe = ∅
    ∧ F ∈ A
```

We're done. Let's solve this.

10.2 VARIABLES: [A → Locs] vs. [Locs → ℘ A], for you and TLC

Our variables are obvious: just a location function

```
l ∈ [A → Locs]
```

I even said so explicitly in the previous section, and you know what I always say: "let's copy-and paste things and move on to the next question without thinking too much about it!"... I never say that, actually. Let's think.

`l ∈ [A → Locs]` is the natural encoding of the situation, when we think abstractly, but is it simple to *use*?

We're going to write a `Next` soon, and we'll need to manipulate "the set of actors on location `i`". How do we compute that, in that case? The function is going the wrong way, what we need is something like `L ∈ [Locs → ℘ A]`, so we can just write `L[i]`. This is actually basically the idea of "let's have 3/N sets of actors", but with the sets conveniently packed in a function rather than manually indexed.

You can get `L` from `l`, of course:

$$L[i] \;=\; l^{-1}[i] \;=\; \{\, a \mid l[a] = i \,\}\,.$$

$l^{-1}[i]$ is called the *preimage of* i.

More generally, in mathematics, for all functions f, the preimage of f is defined defined for all sets S as

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

This is not to be confused with the *inverse function* f^{-1}, which only exists if f is bijective. Mathematics usually distinguishes the two by using $(\cdot)$ for images and $[\cdot]$ for image sets and preimage (sets), but since we're in TLA$^+$, which uses $[\cdot]$ for images, the usual convention is hard to apply, here.

Anyway, the bottom line is that we could derive `L` to write `Next` more easily. Or could we? We'll not just need to compute `L[i]`, when writing `Next`, we'll need to *update* our location function, whether `l` or `L`, keeping it mostly intact but changing the locations of some actors.

There is a syntax for that sort of thing in TLA$^+$:

```
[f ‖ ![x] = e1,
     ![y] = e2, ...]
```

denotes a function `g` that is almost the same as `f`, except that `g[x] = e1 ∧ g[y] = e2`. Furthermore, you can use `@` in the expressions to denote the previous value: `f[x]` in `e1`, `f[y]` in `e2`, and so on.

So which location function is the simplest to manipulate? With `l` we'll override two actors. With `L` we'll override two locations. Equality. The image type for `l` is simpler (`Locs` vs `℘ A`), so *minor* point for `l`.

What happens if we generalise the problem a bit more, with a larger boat, and `K` actors' locations need to be updated? Then `l` breaks down completely, because the `‖` syntax does not scale. We'd need something else entirely (`▶` and `[x ∈ S ↦ e]`, which we'll see later).

Between that and the preimage thing, `L` is clearly the easiest one to actually work with given the tools we have.

But couldn't we have our cake and eat it too, by choosing `VARIABLE l`, deriving `L` from it, and manipulating the state though `L'` instead of `l'` whenever convenient?

Discarding that that would be a lot of work for not much, *yes*, from a purely mathematical point of view, we *could*, but remember that we need to write maths that TLC can actually *compute*. If we write a statement about `L'`, we would be expecting TLC to translate that into a statement about `l'`. If you think "well that's trivial", consider that, with what we have written so far, that's just not possible. There is simply no bijection between `[A → Locs]` and `[Locs → ℘ A]`.

It's possible in our head because we understand that they represent the same thing, and so we have implicit assumptions about `L` we have not yet written down: each actor is on exactly one location. We'll formalise that shortly in `Type`.

Let's pretend we have that all written down; we are expecting TLC to fetch those assumptions from `Type` or somewhere, build a bijection from `[Locs → ℘ A]` to `[A → Locs]`, *prove*, using those assumptions, that it is indeed a bijection, and compute `l'` all on its own.

Not only is TLC not going to *attempt* to do that, but that's exactly the kind of thing which we know *cannot* be automated, be it for humans or computers, or super quantum-computer clusters from the future.[1]

If your variable is `l`, you must have a clear `l' = ...` (not even `... = l'`) somewhere in your `Next`, or TLC is lost.

So we choose the variable type we actually want to work with: the winner is `[Locs → ℘ A]` by a landslide. Now that we have finished that comparison, we'll reclaim the identifier `l` for our variable, as originally planned

While we are at it, we should write our typing invariant, with those hidden assumptions we were talking about earlier:

```
VARIABLES l
Type ≜
    ∧ l ∈ [Locs → ℘ A]
    ∧ ∀ x,y ∈ Locs: x ≠ y ⇒ l[x] ∩ l[y] = ∅
    ∧ ⋃{ l[x] : x ∈ Locs } = A
```

10.3 Fifty shades of `Init`: writing and altering functions

There are many ways to write `Init`. For our convenience, we'll handle the case `N=3`, specifically, in many of the instances presented.

This subsection's main goal is actually to use `Init` as a pretext to study how to write functions.

10.3.1 TLC doesn't know what you know

We have `Locs = 1..3`. Let's say that we start in 1 and want to go to 3. With our choice of variables we basically want to say that

```
l[1] = A ∧ l[2] = ∅ ∧ l[3] = ∅
```

Mathematically it works, but TLC won't like that. It will spit an error message to the effect that it can't compute `l[1]` because he doesn't know what `l` means.

Understand that the poor thing is trying to build a compatible `l` from scratch to get rolling, but, unlike us, it doesn't understand that the above completely defines `l`.

[1] Undecidable problem in computational complexity theory. See Halting Problem and such.

It doesn't actually know that that `l ∈ [Locs → ℘ A]` even though we wrote that in `Type` because,... `Type` might not be an invariant, for one thing; it's something we *check*, not something we *assume*. Assuming we even check it, we might have unchecked the box. And even if we could take it as assumption, TLC would, again, need to invent and prove a theorem to extract the relevant information from the formula.

That's just not doable. So for `Init`, you'll need to write something of the form `l = ...` to get things going. It's easier in the `Next`, because then `l` is already built, and it is trying to build a `l'`, so then that kind of restriction applies to `l'`, but not `l`.

10.3.2 No convenient ↦ notation outside of records

Let's move on; in maths, we could write

```
l = [1 ↦ A, 2 ↦ ∅, 3 ↦ ∅]
```

but TLA$^+$ doesn't accept that syntax; it's reserved for records. There is a good reason for that design choice. Don't expect me to tell you, I haven't the foggiest idea what it might be, but I'm sure there is one, probably.

10.3.3 Introduction to the hidden gems ↦ (:>) and ▶(@@)

Moving on. We can achieve something much like the above by using the fairly obscure, but quite useful operators `:>` and `@@` — which I'm going to render as ↦ [m] and ▶ in this document, I'll explain why shortly — defined (hidden?) in the `TLC` module.

```
l = 1↦A ▶ 2↦∅ ▶ 3↦∅
```

Intuitively, `x↦y` is the function that to `x` associates `y`. That's it. Its whole domain is `{x}`. So under the usual (in mathematics) representation of functions as sets of pairs, `x↦y` = $\{(x, y)\}$. That doesn't hold in TLA$^+$, of course, because functions are primitive, not tuples, but it's still helpful to understand.

As for ▶, it combines two functions into one. Basically, it does the union of functions-as-sets. So `x↦y ▶ X↦Y` means $\{(x, y), (X, Y)\}$. There is a twist, in that in case of conflict between `f` and `g`, `f▶g` resolves it by preferring `f`. Hence the notation I chose [n]: `f` has higher priority than `g`, so $f \geqslant g$, so `f▶g`. That makes sense to me. You can also see `f▶g` as "try `f`, and if you don't find the right value, follow the arrow and try `g`". I read `f▶g` as "`f` overrides `g`". ▶ is associative.

[m] |-> :> render as↦ ↦, to differentiate them.

[n] There is no standard notation for those operators in mathematics, and no traditional maths symbol for them in TLA$^+$, — despite them being very useful — so don't expect to see those notations outside of this document.

10.3.4 The not-quite λ-expression $[x \in S \mapsto e]$

Before seeing how ↦ and ▶ are actually defined in the `TLC` module, we need to introduce another notation for functions. It's the last one, I promise. Except there is another just after, actually. It's the penultimate one, I promise![o]

```
[x ∈ S ↦ e]
```

Note: the ∈ is part of the syntax, for instance, you can't replace "∈ ℘ X" by "⊆ X", even though they are logically equivalent. Same thing for ∀ ∃.

This denotes the function that, to each `x ∈ S`, associates the value denoted by `e`. Of course `e` is an expression that can and likely will contain `x` somewhere. It's basically a lambda expression $\lambda x.\, e$ — in Python, `lambda x: e` — but bounded to the domain `S`. Except there also are `LAMBDA` expressions for *operators* in TLA$^+$, since version 2, so I'm not going to use that terminology for `[x ∈ S ↦ e]`. The definition

```
f ≜ [x ∈ S ↦ e]
```

can equivalently be written

```
f[x ∈ S] ≜ e
```

in which case the identifier `f` can be used in `e` to create recursive functions. We'll do that in another exercise.

Getting back to our operators, we can now understand their definitions:

```
d ↦ e ≜ [x ∈ {d} ↦ e]

f ▶ g ≜ [x ∈ (DOMAIN f) ∪ (DOMAIN g) ↦
           IF x ∈ DOMAIN f THEN f[x] ELSE g[x]]
```

10.3.5 IF THEN ELSE

Wait a minute, that's an `IF THEN ELSE` control structure in the definition of `f ▶ g`. We haven't seen that yet. Noting that it's an *expression*, not a *statement*, as the `x if C else y` construct in Python, because... there is no such thing as a statement in TLA$^+$ or maths,... it does what it says on the tin. There is no `IF THEN` version.

That suggests another way of writing `Init`, though:

```
l = [x ∈ Locs ↦ IF x=1 THEN A ELSE ∅]
```

It works and, unlike previous solutions, is fully general for any `N`.

[o] I lied.

That syntax will clearly be necessary at some point, but for this problem and `N=3`, there are simpler ways of writing this which I want to explore.

10.3.6 CASE ... OTHER

While we're there, let's also showcase

```
CASE P1 → e1
  □ P2 → e2
  ...
  □ OTHER → e0
```

Everything after the first case is optional. We can write `Init` as

```
l = [x ∈ Locs ↦ CASE x=1 → A □ OTHER → ∅]
```

Again, it works, it'll be better than chaining **IF THEN ELSE IF THEN ELSE** in more complicated cases, but it's overkill for our current problem. Also note that the semantics of **CASE** is defined in terms of ↄ, so if several branches *could* be active, a fixed but arbitrary one will be chosen. Don't expect to make a non-deterministic choice with this. And if your reaction is not "well *obviously* it wouldn't even make sense to expect that, because **IF** and **CASE** are expressions denoting values, and there are no such things as non-deterministic values", then you have not read Sec. 7.5.2[p41]: "ↄ vs. ∃: understanding non-determinism" carefully enough. Go read it again.

10.3.7 The winner: tuples and sequences

Back on track, we now can write *and fully understand*

```
l = 1↦A ▸ 2↦∅ ▸ 3↦∅
```

and TLC will accept that without protest. That's fairly simple, clear, and compact. But we can still do better.

After all, what we have boils down to a sequence of sets: $A, \emptyset, \emptyset$.

How are sequences defined in maths? As functions from $[\![1, K]\!]$ to something.

Is there a *Sequence* type in TLA$^+$? You bet. (Tuples too, it's basically the same thing.)

Is it defined as in maths? Absolutely. It indexes from 1, as is common in mathematics, not 0, which is more usual in computer science.

Is there a nice notation for sequences/tuples? Yep, `⟨a,b,c,...⟩`.

So can we just write

```
Init ≜
    l = ⟨A,∅,∅⟩
```

Yes, we can. And will.

Have we just spent however long that was seeing six different versions of `Init`,not counting that one (only four of which actually work!) just to write the *seven characters* ⟨A,∅,∅⟩?

Yes. Yes we did.

. . . But now we are armed, not just with an `Init`, but with a better understanding of what TLC will and will not accept and why, and pretty much every bit of syntax we'll need to write more difficult functions in the next exercises.

Note that our final choice is specific to `N=3`, which might become a problem. But it's very short and easy to read.

---

11 LAB CLASS: Three_Islands

Here is the template you'll complete, with all the code code we settled for during the lecture:

```
---------------------- MODULE Three_Islands --------------------------------
EXTENDS Naturals, TLC

(***************************************************************************)
(* PROBLEM PARAMETERS / INPUTS                                             *)
\* N ∈ ℕ:
\*     number of islands
\* F:
\*     the farmer / boatman
\* Safe:
\*     set of actors who are safe, i.e. have no conflict
\* Unsafe:
\*     set of sets of actors who cannot stay together on an island
(***************************************************************************)

N      ≜ 3
F      ≜ "F"
Safe   ≜ {F}
Unsafe ≜ { {"W1","G"}, {"W2","G"}, {"G","C"} }

(***************************************************************************)
(* DERIVED INPUTS,  HELPER FUNCTIONS, AND INPUT TYPING                     *)
(***************************************************************************)

Locs ≜ 1..N
A ≜ Safe ∪ ⋃Unsafe

ASSUME
    ∧ N ∈ ℕ \ {0}
    ∧ Safe ∩ ⋃Unsafe = ∅
```

```
    ∧ F ∈ A

(*************************************************************************)
(* STATE VARIABLES                                                       *)
\* l ∈ [Locs → ℘ A]:
\*      location ↦ set of actors there
(*************************************************************************)

VARIABLES l
Type ≜
    ∧ l ∈ [Locs → ℘ A]
    ∧ ∀ x,y ∈ Locs: x ≠ y ⇒ l[x] ∩ l[y] = ∅
    ∧ ⋃{ l[x] : x ∈ Locs } = A

(*************************************************************************)
(* INITIAL STATE                                                         *)
(*************************************************************************)

Init ≜
    l = ⟨A,∅,∅⟩

(*************************************************************************)
(* TRANSITIONS                                                           *)
(*************************************************************************)

Licit(S) ≜ something short and sweet

Next ≜ you tell me

(*************************************************************************)
(* PROPERTIES: INVARIANTS & REACHABILITY TARGETS                         *)
(*************************************************************************)
Inv ≜ fun stuff

Target ≜ William Tell's apple
=============================================================================
```

Your turn to play.

Tip: you should expect 129 states, and a length 10 success trace.

12 Scratch that recursion

Obviously, whenever I give you code for auxiliary functions, like

```
max(S) ≜ ∋ x ∈ S: ∀ y ∈ S: y ≤ x
τ̄(S) ≜ max( {τ[a] : a ∈ S} )
```

unless I explicitly tell you "this is beyond the scope of this course", you are expected to learn how it works and be able to produce similar code in the future.

For instance, you should *want* to try to code a `min` operator to practice ∋, without me telling you to.

And of course `{τ[a] : a ∈ S}` should immediately remind you of comprehensions expressions in Python, and that should inspire you a few obvious exercises, like, writing an operator for the Cartesian product.

Each exercise in this course introduces only a reasonable number of new constructs, so you should take the time to become acquainted with them before rushing to the next exercise. Otherwise, you won't remember they exist when you need them.

Refer to the Sec. 17[p93]: "TLA$^+$ CheatSheet" to see how the syntax is written in ASCII and what it means.

In the exercise, we focus on writing code for auxiliary functions.

12.1 Make a scratchpad for experimentation . . . 78
12.2 Playing with ▶ . . . 78
12.3 Reading the precedence / associativity table . . . 79
12.4 **ASSUME** as **assert**: "unit testing" . . . 80
12.5 A recursive function: `fact` . . . 80
12.6 Cardinality: functions vs. operators . . . 81
12.6.1 The tragedy of Darth Set-Of-All-Sets . . . 82
12.6.2 TLC vs. TLA$^+$ standard modules, **LET IN** . . . 83
12.6.3 **RECURSIVE** operators . . . 85
12.7 A straightforward `sum` . . . 85
12.8 **LAMBDA** and Fold / Reduce . . . 86

12.1 Make a scratchpad for experimentation

It's immediately not clear how to integrate "experiment with functions / syntax" into the verification workflow. There is `Print`, but it's a little awkward to use outside of really trivial cases.

So we're going to take a break from solving problems and focus on writing auxiliary functions for a bit. To do that, let's make a "scratchpad" in which to practice. Create a module

```
---- MODULE scratch ----------------------
EXTENDS Integers, Sequences, FiniteSets, TLC

Eval ≜ "Hello"
=========================================
```

Set *No behaviour spec* in *Model Overview*, and in *Model Checking Results* / *Evaluate Constant Expression*, set `Eval` as the expression to evaluate.

Run the model, and you should see `"Hello"` appear in the *Value* field.

The idea is that whenever we want to experiment, we can do so in the scratchpad, putting whatever we want in `Eval`. My scratchpad is my store of useful, well-tested auxiliary functions, which I can paste into modules that need them, after adapting them if necessary.

There are also the community modules to draw upon:

`https://github.com/tlaplus/CommunityModules/tree/master/modules`

It's not as convenient as working in, say, Python, with the interactive mode and everything, but we'll also do a whole lot less complex programming in TLA$^+$ than in Python.

The point of TLA$^+$ is to catch *design* errors, especially for concurrent systems. The systems are usually fairly simple to specify at the very abstract level at which we work; the complexity arises from the interactions between the sub-systems, as I showed in the first lecture with my "two/three clients in a shop" example.

12.2 Playing with ▶

This being said, let's illustrate the merits of our scratchpad; let's say we have a doubt about the ▶ operator: does it favour the left function, or the right one? Let's quickly test this:

```
Eval ≜ [x ∈ 1..3 ↦ 1] ▶ [x ∈ 1..4 ↦ 2]

⟨1, 1, 1, 2⟩
```

It does favour the left one, as expected.

We can also quickly verify that, although syntactically ▶ is left-associative in TLA$^+$ [p], per its definition it is fully associative:

```
r(i) ≜ [x ∈ i..i+2 ↦ i]
Eval ≜ ⟨ r(1) ▶ (r(2) ▶ r(3)),
         (r(1) ▶  r(2))▶ r(3)⟩

⟨⟨1, 1, 1, 2, 3⟩,
 ⟨1, 1, 1, 2, 3⟩⟩
```

12.3 Reading the precedence / associativity table

Let's say now that we don't like the "overrides / ▶ / @@" operator, and would prefer to work with an "extends / !!" operator [q], such that `f !! g` extends `g` with pairs of `f` not set in `g`.

Turns out, we can make our own infix operators, provided they are on the list of syntax supported for that purpose (see the official Cheat Sheet for that, along with their precedences and associativity). Let's do it:

```
f !! g ≜ g ▶ f
Eval ≜ ⟨r(1) ▶ r(2), r(1) !! r(2)⟩

⟨⟨1, 1, 1, 2⟩,
 ⟨1, 2, 2, 2⟩⟩
```

Now let's try `r(1) !! r(2) !! r(3)`. Oops, we get a precedence conflict.

Looking at the precedence table, we see it has precedence 9-13, and is not left-associative, which is the only associativity proposed by TLA$^+$. In other words, syntactically, it is not associative at all. Hence the message.

Let's go on the hunt for a suitable replacement: we need something associative. `&&` is available and associative and ready and willing... and if you try it, it will work for the above:

```
Eval ≜ r(1) && r(2) && r(3)

⟨1, 2, 3, 3, 3⟩
```

but you'll get a precedence error message for ↦ the moment you use it to write, say

```
Eval ≜  0 ↦ 1 && 1 ↦ 2
```

Let's look at the table again: ↦ has precedence 7-7. && has 13-13. That means that && binds more tightly than ↦. That means that what we wrote really meant `0 ↦ (1 && 1) ↦ 2`, and that we get that message because ↦ is not associative.

[p] See the official Cheat Sheet for a precedence table.

[q] I'm not rendering it as the obvious ◀ because we're not actually going to keep it...

That means we are back on the hunt, for any infix operator syntax that's associative and has precedence strictly less than 7. The only ones that fit the bill are $\wedge$ $\vee$, which are already taken, obviously.

So the conclusion is *there is no suitable infix syntax allowed by TLA$^+$ for our "extends" operator*.

Moral of the story: there is a precedence / associativity table; learn to read and use it. That's good advice for any language, mind you, not just TLA$^+$. You'll need it to debug *something* at some point, I'm sure.

12.4 `ASSUME` as `assert`: "unit testing"

Use the scratch pad as testing grounds for your auxiliary functions. You can use **`ASSUME`** in much the same way I made you use **`assert`** in Python for the purpose of unit testing.

For instance:

```
max(S) ≜ ℑ x ∈ S : ∀ y ∈ S : y ≤ x
min(S) ≜ ℑ x ∈ S : ∀ y ∈ S : y ≥ x

_test_Int_set ≜ {6, 9, 5, -1, 16, 1}
ASSUME
    ∧ max(_test_Int_set) = 16
    ∧ min(_test_Int_set) = -1
```

Validate that **`ASSUME`** actually does something by writing wrong values.

It's not really at all what **`ASSUME`** is *for*; unit tests statements are **`THEOREM`**s, not assumptions, but it's convenient, and we'll not cover cases where the distinction matters[(r)], so we'll do it.

12.5 A recursive function: `fact`

TLA$^+$ is maths; that means there is no **`while`** or **`for`** loop. We still want to compute sums and products, though, so we'll use recursion.

Let's play with factorial:

```
fact[n ∈ ℕ] ≜ IF n = 0 THEN 1 ELSE n * fact[n-1]

ASSUME
    ∧ fact[5] = 120
    ∧ fact ∈ [ℕ→ℕ]
    ∧ DOMAIN fact = ℕ
```

[(r)]TLAPS... writing machine-checked proofs in TLA$^+$.

It works as expected, yet you will find that TLC doesn't like `fact ∈ [ℕ → ℕ]`. The error message should be

```
Attempted to compute the number of elements in the overridden value Nat.
```

It's not too surprising that TLC has a problem somewhere, ℕ is infinite, the computer is finite, but then why does it have a problem only with `fact ∈ [ℕ → ℕ]` and not also with `DOMAIN fact = ℕ`?

Well, the latter can be checked symbolically, from the definition of `fact[n ∈ ℕ]`, which includes the domain. You don't need to enumerate ℕ to infer that ℕ = ℕ.

Note that this doesn't mean that `fact` is *well-defined* over its domain. Replacing the recursive call with `fact[n-10]` will not violate `DOMAIN fact = ℕ` despite `fact` being incorrectly defined for `n < 10`. You can elide that if you're just checking the domain of *definition*. There *is* a symbolic definition. Not a good one, but it's there.

However, with `fact ∈ [ℕ → ℕ]`, you have to verify that each input value to `fact` does produce an output value in ℕ. You won't do that any time soon by checking them all individually. You need to write a proof. It's a trivial proof by induction in that case, but doing so is well outside the scope of TLC.

Again, writing proofs *cannot* be automated, whether by machines or humans. Even if TLC came packaged with a chibi mathematician in a little cage, it would still fail on *some* proofs.

Another limitation is that, despite functions being values, like integers and sets, using `fact` as the value of a variable, as in

```
Init ≜ x = fact
```

will net you the same error message. TLC doesn't like infinite structures, generally.

Yet, you can still *use* `fact` on specific values, obviously.

```
Init ≜ x = fact[5]
```

is *fine*. That's what we defined `fact` for, probably. If we couldn't do that, functions would be useless.

12.6 Cardinality: functions vs. operators

In the `FiniteSets` module is defined the operator `Cardinality`, giving you the cardinality, or number of elements, of any set.

Intuitively, for finite sets it's a trivial recursive definition: $\varnothing$ is 0, and for the inductive case $\{e_1, \ldots, e_n\}$, take out any element of the set, say e_1, do $1+$ recursive call on $\{e_2, \ldots, e_n\}$.

Isolating an element from a set is how induction is done on them, so let's define two neat functions, for all our induction-on-sets needs:

```
peek(S) ≜ ℑ x ∈ S: ⊤
rest(S) ≜ S \ {peek(S)}

ASSUME ∀ S ∈ ℘(1..5) \ {∅}:
         ∧ S = {peek(S)} ∪ rest(S)
         ∧ peek(S) ∉ rest(S)
```

Note that `S = {peek(S)} ∪ rest(S)` holds because ℑ is arbitrary but fixed, as discussed at length in Sec. 7.5.2[p41]: "ℑ vs. ∃: understanding non-determinism".

Let's try defining our own *function* `Card` — as opposed to *operator* — and see own far we can go:

```
Card[S ∈ ....
```

Well, we didn't go very far. Functions have a domain. You can't define them without giving or otherwise explicitly building a domain. If we knew that we only wanted to deal with sets of integers we could write

```
CardZ[S ∈ ℘ℤ] ≜
    IF S = ∅
    THEN 0
    ELSE 1 + CardZ[ S \ {peek(S)} ]

ASSUME ∀ S ∈ ℘(1..5): CardZ[S] = Cardinality(S)
```

and that would work,... but then do we need another function for sets of strings? Theoretically `CardZ[S ∈ ℘(ℤ ∪ STRING)]` would work for both (TLC won't like that for technical reasons), but then what if I want anything else?

What's the domain of `Card`, *really*? Well, any set has a cardinality, so... the set of all sets. Let's call it $\mathbb{S}$. Feelin' good.

12.6.1 The tragedy of Darth Set-Of-All-Sets

So what's the TLA$^+$ for $\mathbb{S}$?

Weeeeell... let's talk about this. If S is a set, then $S \in S$ has a truth value. Usually false, presumably, for any example that comes readily to mind but clearly $\mathbb{S} \in \mathbb{S}$. And it's not the only one, there are infinitely many.

Let's consider a finite set S, and take $\overline{S} = \mathbb{S} \setminus \{S\}$. This is a set, because we just said $\mathbb{S}$ is a set: $\overline{S} \in \mathbb{S}$.

We have $S \neq \overline{S}$, because one is finite and the other not, and thus $\overline{S} \notin \{S\}$, which means that $\overline{S} \in \mathbb{S} \setminus \{S\} = \overline{S}$. That shows that there are plenty of sets that are members of themselves, and plenty that are not.

All the sets we deal with normally are not members of themselves: $[\![1, 10]\!]$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$,... those are the *normal* ones ($S \notin S$), the others are *abnormal* ($S \in S$). We should probably distinguish the two, because the abnormal ones seem pretty icky. We want to work only with normal sets.

So let's define R (for Bertrand Russell) the set of normal sets:

$$R \triangleq \{S \in \mathbb{S} \mid S \notin S\}$$

Problem solved. Have a cookie.

Before dipping the cookies in milk, let's check that R is normal: that is to say that it belongs to the set of normal sets R, that is to say $R \in R$; that is to say that it is abnormal; that is to say, that $R \notin R$,... Oh.

$$R \in R \iff R \notin R\,.$$

Houston, we have a problem.

This is known as Russell's Paradox. It's kind of a big deal in mathematics. For this and other reasons (Cantor's Paradox: $|2^{\mathbb{S}}| > |\mathbb{S}|$, yet $2^{\mathbb{S}} \subset \mathbb{S}$, so $|2^{\mathbb{S}}| < |S|$, ...) there is no set of all sets. Mathematicians call something like $\mathbb{S}$ a *collection*, instead. You can't manipulate it as a set. It can't be used as the domain of a function.

Anything at least as big as $\mathbb{S}$ (surjection onto $\mathbb{S}$) is a collection.

So... `Card` cannot be defined as a function.

12.6.2 TLC vs. TLA$^+$ standard modules, `LET IN`

If `Card` cannot be defined as a function, how the hell did the `FiniteSets` module define it? They defined it as an *operator* which, being just a syntactic definition, doesn't have a domain:

```
Cardinality(S) ≜
    LET CS[T ∈ ℘ S] ≜
        IF T = ∅
        THEN 0
        ELSE 1 + CS[T \ {Ↄ x : x ∈ T}]
    IN  CS[S]
```

There is much to unpack, here. If you try to use that definition yourself, you'll get an error:

`TLC attempted to evaluate an unbounded CHOOSE. Make sure that the expression is of form Ↄ x ∈ S: P(x).`

That's what I told you earlier about Ↄ. So what's going on? The definition above comes straight from `FiniteSet` a standard module of TLA$^+$. Is it wrong?

No, but TLC can't run it. TLA$^+$ is not a programming language, and TLA$^+$ is not TLC. TLA$^+$ is a mathematical formalism. TLA$^+$is "a general-purpose formal specification language that is particularly useful for describing concurrent and distributed systems".

You can use it on paper, like maths, because it *is* maths. You can use it in a variety of tools, including TLAPS (TLA$^+$ Proof System), which is a tool to write machine-checked proofs. The mathematician chooses each proof step, and the machine checks that each step is valid. Then there are model-checkers, like TLC and Apalache, which do very different things. TLC builds the state machine explicitly, Apalache builds it symbolically. Each tool built around TLA$^+$ has its own limitations as to which subset of valid TLA$^+$ constructs it can handle. We happen to use TLC, so we focus on its restrictions.

But... wait, wait. TLC can't use the definition above, but... I can `EXTENDS FiniteSets` and use `Cardinality` without problem. What's going on?

TLC doesn't actually use the definition above. Nor does it use the TLA$^+$ definition of $\mathbb{N}$ based on Peano's axioms (remember that 4A FISA, from last year, with me?). It is implemented in Java, so it *overrides* most mathematical definitions with the corresponding Java objects and methods.

Your own operators don't enjoy that special treatment, though, so we'll have to write that in a way that TLC can understand, which is what I did in `peek`. `∋ x ∈ S: ⊤` is the same thing as `∋ x: x ∈ S`, except TLC can read it.

With this, we can at last define our `Card`, and check that it works as expected by comparing it to the reference implementation `Cardinality`:

```
Card(S) ≜
    LET CS[T ∈ ℘ S] ≜
        IF T = ∅
        THEN 0
        ELSE 1 + CS[T \ {peek(T)}]
    IN  CS[S]

ASSUME ∀ S ∈ ℘ (1..5): Card(S) = Cardinality(S)
```

It works, neat.

Wait, we still don't understand what `LET IN` is, or *why* it works.

`LET x ≜ e IN e2` is just a local definition. It's just like any other definition `x ≜ e` in the module, except it only exists within the scope `e2`. That's it.

The interesting thing here is that, while the collection of all sets $\mathbb{S}$ is too big to be a domain, once you pick a specific set $S \in \mathbb{S}$, $\wp(S)$ is a valid domain. So you can write a recursive function `CS` of domain $\wp(S)$, no problem.

If this feels like a trick, and you are asking "why do this instead of defining a recursive *operator* `Card`?", then you're quite right.

The main reason I show you this pattern is that it is used in almost all TLA$^+$ libraries you'll find. The reason it is widely used is that the first version of TLA$^+$ did not support recursive operators at all. The author, Leslie Lamport [(s)] just didn't know how to define them properly. For instance, he wondered what to do with this kind of silly definition:

```
F ≜ ∋ v : v ≠ F
```

Again I remind you that TLA$^+$ is not a programming language. Implementing recursive calls by a stack and a jump assembly instruction is one thing. Building a general recursive mathematical formalism without running into fun things like Russell's Paradox is an entirely different kettle of fish.

Eventually, he figured it out, and the current version, TLA^{+2}, does support it. (But only for first-order operators, more on that later.).

12.6.3 RECURSIVE operators

The definition of a recursive operator must be preceded by a **RECURSIVE** declaration. You must pass the arity of the operator: `(_)` for one argument, `(_,_)` for two (works for prefix and infix operators), etc.

We obtain:

```
RECURSIVE Card(_)
Card(S) ≜
    IF S = ∅
    THEN 0
    ELSE 1 + Card( S \ {peek(S)} )

ASSUME ∀ S ∈ ℘(1..5): Card(S) = Cardinality(S)
```

And it works.

To recap, we'll use

(1) recursive functions if there is a clear domain,

(2) recursive operators if there is not.

12.7 A straightforward sum

With this, we can implement, for instance, the sum of all elements of a set, without presuming the exact numeric type of it.

This is what I really wanted, actually, we'll need it for the next exercise.

[(s)] The La in LaTeX, among other things...

```
RECURSIVE sum(_)
sum(S) ≜
    IF S = ∅
    THEN 0
    ELSE peek(S) + sum(rest(S))

ASSUME ∀ n ∈ 0..6: sum(0..n) = ( n*(n+1) ) ÷ 2
```

12.8 LAMBDA and Fold / Reduce

If you have done some functional programming (OCaml, Haskell, Lisp, ...) you know that `sum` is a special case of the higher-order function commonly called `fold`, or `reduce`.

Another neat thing introduced in TLA^{+2} is **LAMBDA**. It works exactly as **lambda** in Python, and defines anonymous operators.

```
LAMBDA x,y: x ∪ y
```

What's it good for? Passing to higher-order operators, that is to say operators that take other operators as inputs. For instance

```
FoldMap(S, e, op(_,_), f(_)) ≜ ...
```

is the signature for folding a set `S` on a base vale `e`, with a binary operator `op`, applying `f` to every element. With this, we could define `sum` just as

```
FoldMap(0..n, 0, +, LAMBDA x:2*x)
```

Can we implement a general fold operator on sets?

```
RECURSIVE FoldMap(_,_,_,_)
FoldMapSet(S, e, op(_,_), f(_)) ≜ ...
```

will unfortunately not work. Not only will TLC complain about the arity, but the meaning of recursive higher-order operators remains undefined even in TLA^{+2}.

We will have to use a recursive sub-function[(t)]:

```
FoldMap(S, e, op(_,_), f(_)) ≜
    LET z[s ∈ ℘ S] ≜
        IF s = ∅
        THEN f(e)
        ELSE op( f(peek(s)), z[rest(s)] )
    IN z[S]

Fold(S, e, op(_,_)) ≜ FoldMap(S, e, op, LAMBDA x:x)
```

[(t)] If you had and remember the Python class with me, you know I always name my recursive sub-functions z.

```
Reduce(S, op(_,_)) ≜ Fold(rest(S), peek(S), op)

ASSUME ∀ n ∈ 0..6: FoldMap(0..n, 0, +, LAMBDA x:2*x) = n*(n+1)
ASSUME ∀ n ∈ 0..6: Reduce(0..n, +) = ( n*(n+1) ) ÷ 2
```

This works.

---

13 LAB CLASS: Scratchpad

Define the following functions

(1) `IMAGE(f)` is the image set of a function

```
IMAGE(f) ≜ ???

ASSUME IMAGE([x ∈ 0..2 ↦ 2*x]) = {0, 2, 4}
```

(2) `PREIMAGE(f)` is the functions which to each element of the image of `f` associates its preimage, which is to say the set of its antecedents.

```
PREIMAGE(f) ≜ ???

ASSUME PREIMAGE([x ∈ 0..2 ↦ 2*x]) = [x ∈ {0,2,4} ↦ {x ÷ 2}]
```

(3) `argmax(f,S)` is the value of `x ∈ S` that maximises the function `f`.

```
argmax(f,S) ≜ ???

ASSUME
    LET f ≜ [x ∈ ℤ↦ x^2] IN
    ∀ S ∈ ℘(-2..2) \ {∅}: f[argmax(f,S)] = max( { f[x] : x ∈ S } )
```

(4) `setto(S,v,f)` is the function `f`, altered so that it associates `v` to every element of `S`.

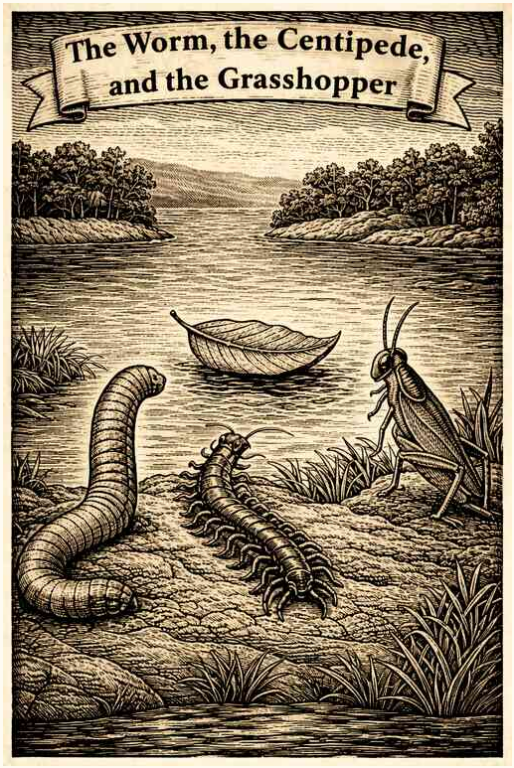
Say, for instance, a location function, with a set of actors changing locations...

```
setto(S,v,f) ≜ ???

ASSUME setto(2..3, 0, ⟨1, 2, 3, 4, 5⟩) = ⟨1, 0, 0, 4, 5⟩
```

(5) `prod(S)` is the product $\prod_{x \in S} x$.

```
prod(S) ≜ ???

ASSUME ∀ n ∈ 1..6: prod(1..n) = fact[n]
```

14 LAB CLASS: The Worm, the Centipede, and the Grasshopper



Let's have a fresh new problem:

> *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, how fast can this be achieved?*

You'll solve this in its obvious generality, and instantiate the problem parameters for the above.

> *Let* $A = \{ B, a_1, \ldots, a_n \}$ *be a set of "actors" starting on the left bank. They all wish to cross to the right bank. All have nonnegative weights, given by the weight function* $w : A \to \mathbb{N}$. *The boat* B *cannot operate itself, but any other 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.*

I'm not giving you any TLA$^+$ code this time. You have all the tools you need now.

You will *not* duplicate the logic for left-right and right-left moves.

Tip: you should expect 14 distinct states, and a success trace of length 6.

15 LAB CLASS: Hard Mode: One Spec To Cross Them ALL!

Disclaimer: *this exercise is **completely optional**, and just there as a safety valve to ensure the stronger students don't run the risk of running out of stuff to do. Don't you hate it when that happens? Proceed if (and only if) that's the case for you.*

To help me quickly test exercise ideas, I wrote a very general specification for river-crossing problems; all problems we have seen so far — and many more besides — are particular instances of that.

Here is the raw documentation for the input — not very rigorous because written purely for my own benefit, I didn't intend to make an exercise out of it:

```
(*****************************************************************************
User Inputs:

Topology: set of sets of inter-navigable locations.
    e.g. { {1,2}, {2,3,4} }: one can go 1<↔2 and 2<↔3<↔4.

ILocs: location → set of actors there (boats and passengers)
    Initial location assignment of actors.
    Note that a boat may become the passenger of another boat,
    if not explicitly forbidden in LicitB.

Licit(S): S is a licit set of actors in any location

LicitB(S): S is a licit set of actors on a boat

PassengerW: passenger actor → weight ∈ Nat
    Any actor not included here will have weight 0

BoatCWO: boat actor → ⟨Capacity, Max Weight, Operators (set of passengers)⟩
    Capacity: how many actors may I carry within myself
    Max W: total weight of actors carried, myself included
    Operators: who can operate me; a boat may be its own operator.
        A boat can only move with an operator.

Budget: initial move budget ∈ Nat

MoveCost(from,to,S): how much does it cost to move S ∈ Actors
    from one location to another

Dest: on which location do we want everybody to be?
*****************************************************************************)
```

And here are how the previous problems are encoded. Operators that are not inputs, but part of the problem's vocabulary, are by convention prefixed by _.

```
(* Wolf, Goat, Cabbage *)
_A ≜ {"W", "C", "G", "F"}
Topology ≜ { 1..2 }
ILocs ≜ ⟨_A, ∅⟩
_ProbGroups ≜ { {"W","G"}, {"G","C"} }
Licit(S) ≜ (∃ G ∈ _ProbGroups: G ⊆ S) ⇒ "F" ∈ S
LicitB(S) ≜ ⊤
BoatCWO ≜ [F ↦ ⟨1, 0, {"F"}⟩]
PassengerW ≜ null
Budget ≜ 0  MoveCost(from,to,S) ≜ 0
Dest ≜ 2

(* Indiana Jones *)
_time ≜ [ I ↦ 1, G ↦ 2, W ↦ 4, K ↦ 8, L ↦ 0 ]
_A ≜ DOMAIN _time
Topology ≜ { 1..2 }
ILocs ≜ ⟨_A, ∅⟩
Licit(S) ≜ ⊤  LicitB(S) ≜ ⊤
BoatCWO ≜ [L ↦ ⟨2, 0, _A \ {"L"}⟩]
PassengerW ≜ null
Budget ≜ 15
MoveCost(from,to,S) ≜ maxf(_time,S)
Dest ≜ 2

(* 2 Wolves, Goat, Cabbage, 3 Islands *)
_A ≜ {"W1", "W2", "C", "G", "F"}
Topology ≜ { 1..3 }
ILocs ≜  ⟨_A,∅,∅⟩
_ProbGroups ≜ { {"W1","G"}, {"W2","G"}, {"G","C"} }
Licit(S) ≜ (∃ G ∈ _ProbGroups: G ⊆ S) ⇒ "F" ∈ S
LicitB(S) ≜ ⊤
BoatCWO ≜ [F ↦ ⟨1, 0, {"F"}⟩]
PassengerW ≜ allcst(_A, 0)
Budget ≜ 0  MoveCost(from,to,S) ≜ 0
Dest ≜ 3

(* Worm, Centipede, and Grasshopper *)
_A ≜ {"W", "C", "G", "B"}
Topology ≜ { {"l", "r"} }
ILocs ≜  [l ↦ _A, r ↦ ∅]
Licit(S) ≜ ⊤  LicitB(S) ≜ ⊤
BoatCWO ≜ [B ↦ ⟨3, 60, _A \ {"B"}⟩]
PassengerW ≜ [W ↦ 50, C ↦ 30, G ↦ 20]
Budget ≜ 0  MoveCost(from,to,S) ≜ 0
Dest ≜ "r"
```

If you have done all other exercises and you're bored, solve *this* :-)

16 LAB CLASS: Toggle Problems

We're done with river-crossing problems. Let's shake things up a little with what I call "toggle problems".

This is a kind of puzzle often found in video games[u].

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.*

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.

Of course, we'll generalise this to all possible problems of this form:

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

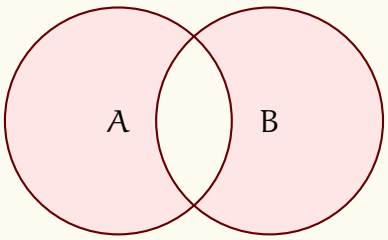
Get to work :-)

On the given instance, you should expect 16 distinct states, and a success trace of length 4.

[u] 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.

Tip: *Depending on your choice of variables, you should think about* ▶*, XOR, or the symmetric difference between sets:*



.

Tip: you should expect 16 states and a length 4 success trace.

17 TLA$^+$ CheatSheet

See also the official Cheat Sheet, which I put on Celene.

This one is shamelessly taken from https://mbt.informal.systems/docs/tla_basics_tutorials/tla+cheatsheet.html — and slightly modified.

link book, elements of semantics

```
(* Comments *)

(* This is a
   multiline comment *)
\* This is a single line comment

(* Module structure *)

---- MODULE <module> ----   \* Starts TLA+ module (should be in file <module>.tla)
====                        \* Ends TLA+ module (everything after that is ignored)

EXTENDS <module>            \* EXTEND (import) another TLA+ module
VARIABLES x, y, ...         \* declares variables x, y, ...
CONSTANTS x, y, ...         \* declares constants x, y, ... (should be defined in configuration)

Name == e                   \* defines operator Name without parameters, and with expression e as a body
Name(x, y,...) == e         \* defines operator Name with parameters x, y, ..., and body e (may refer to x, y, ...)

(* Boolean logic *)

BOOLEAN                     \* the set of all booleans (same as {TRUE, FALSE})
TRUE                        \* Boolean true
FALSE                       \* Boolean false
~x                          \* not x; negation
x /\ y                      \* x and y; conjunction (can be also put at line start, in multi-line conjunctions)
x \/ y                      \* x or y; disjunction (can be also put at line start, in multi-line disjunctions)
x = y                       \* x equals y
x /= y                      \* x not equals y
x => y                      \* implication: y is true whenever x is true
x <=> y                     \* equivalence: x is true if and only if y is true

(* (Bounded) Quantifiers *)

\A x \in S: e               \* for all elements x in set S it holds that expression e is true
\E x \in S: e               \* there exists an element x in set S such that expression e is true
CHOOSE x \in S: e           \* choose some element c in set S such that e. Make sure it exists before!

\A x,y \in S, z\in S': e    \* --- example of generalisation to multiple variables

(* Unbounded quantifiers, UNSUPPORTED BY TLC! *)

\A x: e                     \* for all elements x it holds that expression e is true
\E x: e                     \* there exists an element x such that expression e is true
CHOOSE x: e                 \* choose some element c such that e. Make sure it exists before!

(* Integers *)              \* EXTENDS Integers (should extend standard module Integers)

Nat                         \* the set of all natural integers (an infinite set)
Int                         \* the set of all integers (an infinite set)
1, -2, 1234567890           \* integer literals; integers are unbounded
a..b                        \* integer range: all integers between a and b inclusive
x + y, x - y, x * y         \* integer addition, subtraction, multiplication
x \div y                    \* integer division (/ is for Reals)
```

```
x < y, x <= y               \* less than, less than or equal
x > y, x >= y               \* greater than, greater than or equal

(* Strings *)

STRING                      \* the set of all finite strings (an infinite set)
"", "a", "hello, world"     \* string literals (can be compared for equality; otherwise uninterpreted by TLC)

(* Finite sets *)           \* EXTENDS FiniteSets (should extend standard module FiniteSets)

{a, b, c}                   \* set constructor: the set containing a, b, c
Cardinality(S)              \* number of elements in set S
IsFiniteSet(S)              \* is S a finite set?
x \in S                     \* x belongs to set S
x \notin S                  \* x does not belong to set S
S \subseteq T               \* is set S a subset of set T? true of all elements of S belong to T
S \union T                  \* union of sets S and T: all x belonging to S or T
S \cup T                    \* union, also
UNION S                     \* union of all elements of set-of-sets S
SUBSET S                    \* powerset of S: set of all subsets of S
S \intersect T              \* intersection of sets S and T: all x belonging to S and T
S \cap T                    \* intersection, also
S \ T                       \* set difference, S less T: all x belonging to S but not T
{x \in S: P(x)}             \* set filter: selects all elements x in S such that P(x) is true
{e: x \in S}                \* set map: maps all elements x in set S to expression e (which may contain x)

{a,b,c : a,b \in A, c \in C}\* -- example of generalisation to multiple variables

(* Functions *)

[S -> T]                    \* set of all total functions from domain S to T

[x \in S |-> e]             \* function constructor: maps all keys x from set S to expression e (may refer to x)
f[x]                        \* function application: the value of function f at key x
DOMAIN f                    \* function domain: the set of keys of function f
[f EXCEPT ![x] = e]         \* function f with key x remapped to expression e (may reference @, the original f[x])
[f EXCEPT ![x] = e1,        \* function f with multiple keys remapped:
          ![y] = e2, ...]   \*   x to e1 (@ in e1 will be equal to f[x]), y to e2 (@ in e2 will be equal to f[y])
d :> e                      \* EXTENDS TLC: function of domain {d}, mapping d to e
f @@ g                      \* EXTENDS TLC: function extending f with g, preferring f in case of conflict

(* Records *)

[x |-> e1, y |-> e2, ...]   \* record constructor: a record which field x equals to e1, field y equals to e2, ...
r.x                         \* record field access: the value of field x of record r
[r EXCEPT !.x = e]          \* record r with field x remapped to expression e (may reference @, the original r.x)
[r EXCEPT !.x = e1,         \* record r with multiple fields remapped:
          !.y = e2, ...]    \*   x to e1 (@ in e1 is equal to r.x), y to e2 (@ in e2 is equal to r.y)
[x: S, y: T, ...]           \* record set constructor: set of all records with field x from S, field y from T, ...

(* Sequences *)             \* EXTENDS Sequences (should extend standard module Sequences)

<<a, b, c>>                 \* sequence constructor: a sequence containing elements a, b, c
s[i]                        \* the ith element of the sequence s (1-indexed!)
s \o t                      \* the sequences s and t concatenated
Len(s)                      \* the length of sequence s
Append(s, x)                \* the sequence s with x added to the end
Head(s)                     \* the first element of sequence s

(* Tuples *)

<<a, b, c>>                 \* tuple constructor: a tuple of a,b,c (yes! the <<>> constructor is overloaded)
                            \* - sequence elements should be same type; tuple elements may have different types
t[i]                        \* the ith element of the tuple t (1-indexed!)
```

```
S \X T                      \* Cartesian product: set of all tuples <<x, y>>, where x is from S, y is from T

(* State changes *)

x', y'                      \* a primed variable (suffixed with ') denotes variable value in the next state
UNCHANGED <<x,y>>           \* variables x, y are unchanged in the next state (same as x'=x /\ y'=y)

(* Control structures *)

LET x == e1 IN e2           \* introduces a local definition: every occurrence of x in e2 is replaced with e1
LET x==e1 y==e2 ... IN eN   \* --- generalisation
IF P THEN e1 ELSE e2        \* if P is true, then e1 should be true; otherwise e2 should be true
CASE P1 -> e1               \* CHOOSE an applicable value; catch-all case OTHER optional
  [] P2 -> e2
  ...
  [] OTHER -> eO
```

Part III

OLD Lab classes

18 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.

18.1 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.

18.1.1 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.

18.1.2 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.

18.1.2.1 *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.

18.1.2.2 *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.[v]

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

[v] 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.)

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.

18.1.2.3 *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.

18.1.2.4 *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.

18.1.2.5 *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.

18.1.3 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.

19 Basic finite state systems

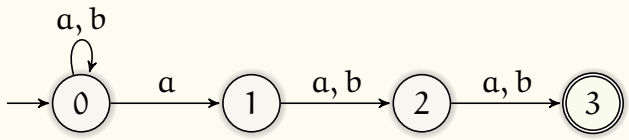
Let's brush up on automata and familiarise ourselves with the NFA framework.

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:

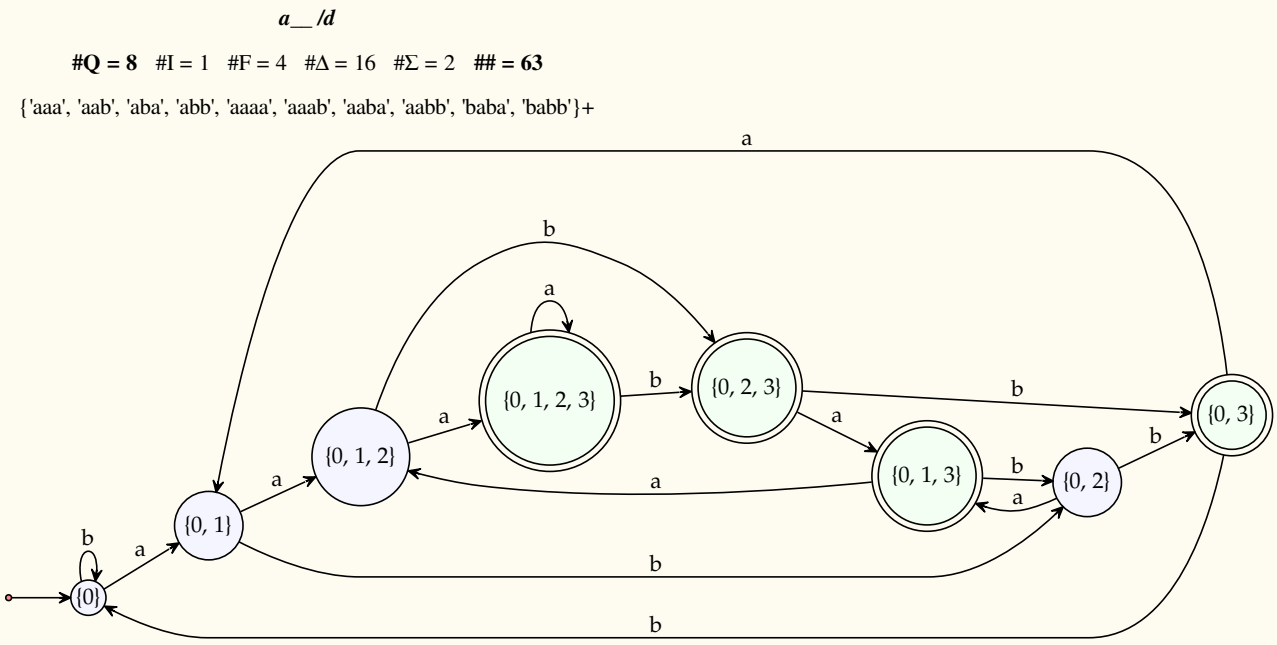
```
A = NFA.spec("""
    0
    3
    0 a 0 b 0 a 1
    1 a 2 b 2
    2 a 3 b 3
    """, name='a__, bis')
```

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:



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:

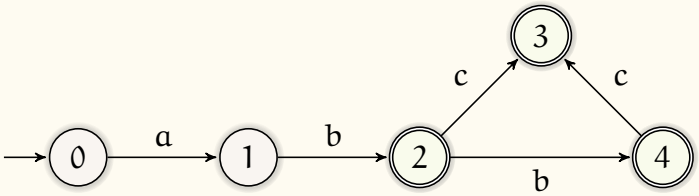
		a	b
Initial	$\{0\}$	$\{0,1\}$	$\{0\}$
	$\{0,1\}$	$\{0,1,2\}$	$\{0,2\}$
	$\{0,2\}$	$\{0,1,3\}$	$\{0,3\}$
Final	$\{0,3\}$	$\{0,1\}$	$\{0\}$
	$\{0,1,2\}$	$\{0,1,2,3\}$	$\{0,2,3\}$
Final	$\{0,1,3\}$	$\{0,1,2\}$	$\{0,2\}$
Final	$\{0,2,3\}$	$\{0,1,3\}$	$\{0,3\}$
Final	$\{0,1,2,3\}$	$\{0,1,2,3\}$	$\{0,2,3\}$

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 $\cup$, `&` for $\cap$, `+` 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

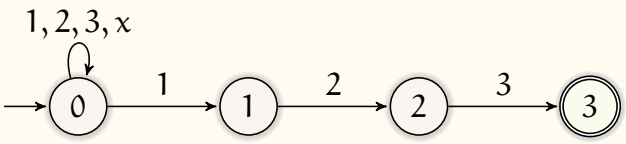
```
NFA.pdf_renderer.print_status()
```

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. 25.4.4[p132]: "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.

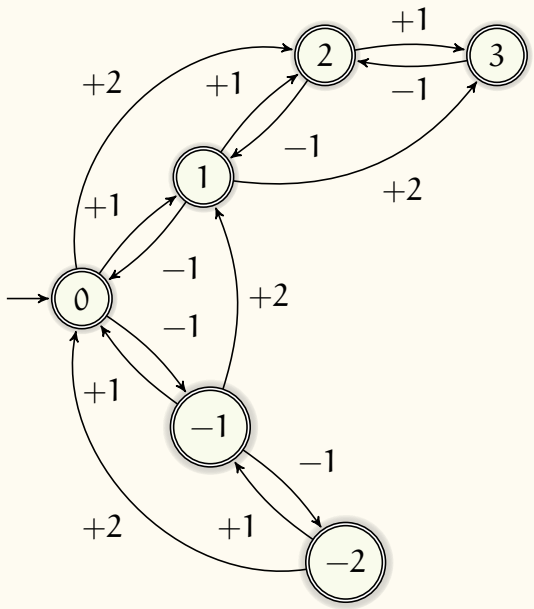
For all $n, m, i \in \mathbb{Z}$ and $X \subseteq \mathbb{Z}$, formally define a NFA $V(n, m, i, X)$ representing an integer variable on the interval $[\![n, m]\!]$, initialised to i, that can be incremented by the quantities in X, and only by those.

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

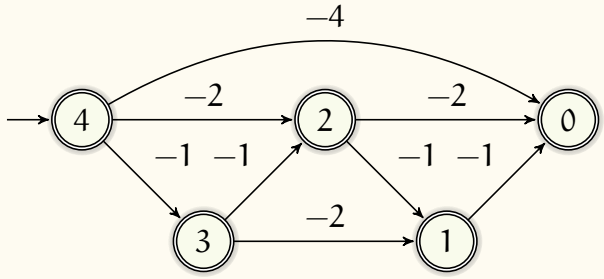
We permit neither overflow nor underflow. We consider all states final.

For instance, we have:

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

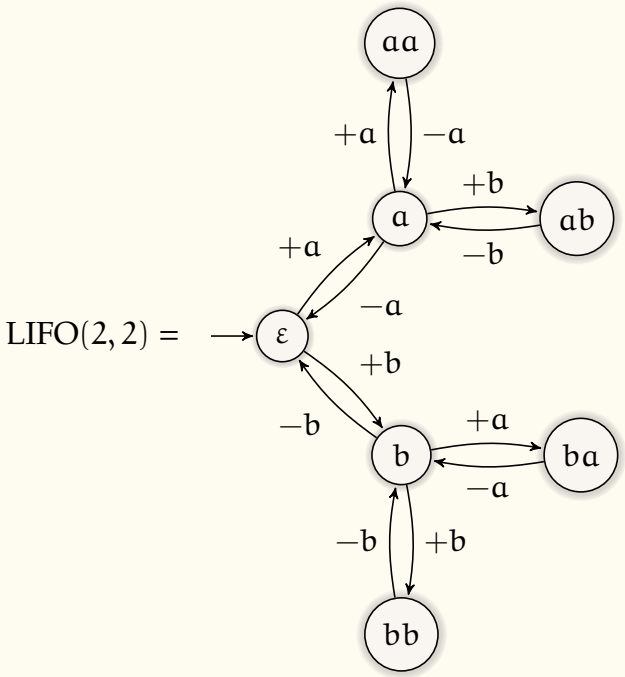


$V(0, 4, 4, \{-1, -2, -4\})$ =



(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.



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. 25.4.4[p132]: "Incrementable Integer Variable". Try to do that on your own, then you can check the solution given in the lecture notes: Sec. 27.0.1[p135]: "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:

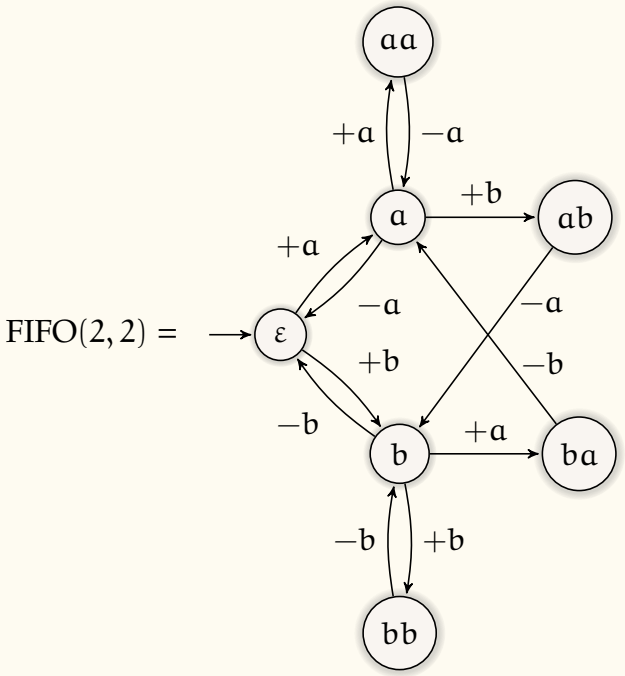
```
A = NFA({""},{}, { }, name=f"LIFO({n} symbs, {m} cap)" )

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

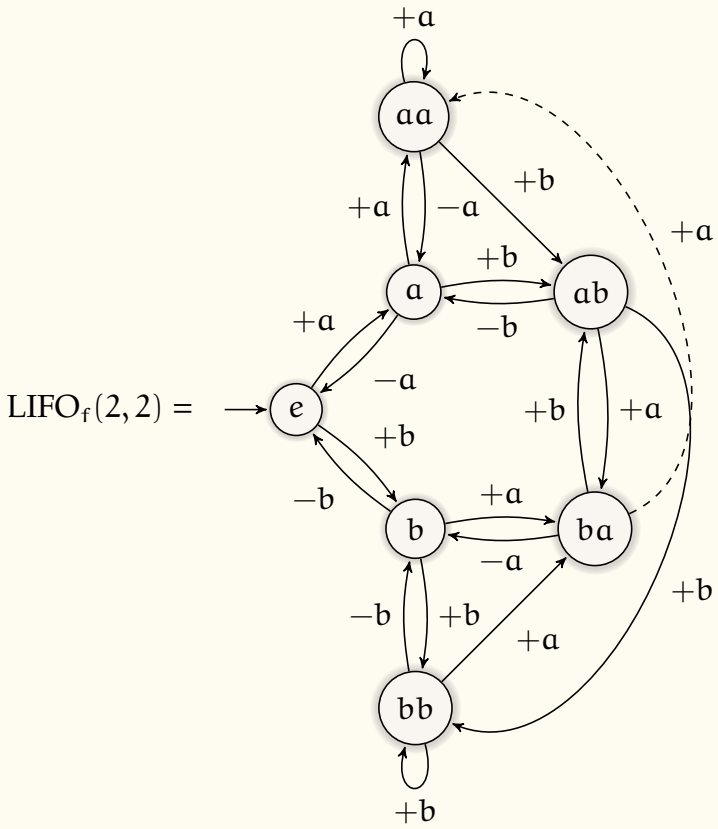
generated automaton to visualise, step by step, which states and transitions were added; which is pretty neat, if I do say so myself ;-)

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 **i**n, **f**irst **o**ut) queue on n distinct symbols, of capacity m.



(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.



(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:

$$\varepsilon, [], [][], [[]], [[][]], [[]][], [][[]], [[[]]], [][][], [[[][]]], \dots$$

Here are a few words that are not Dyck words:

$$], [,][,]], [[,]]], [],][,][[, [[, [[[,]][, [[,]]],]]][,]][[,][[[, \dots$$

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. 25.4.4[p132]: "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,"()[]{}<>")`[w].

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. 27.1.2[p142]: "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.

[w] 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:

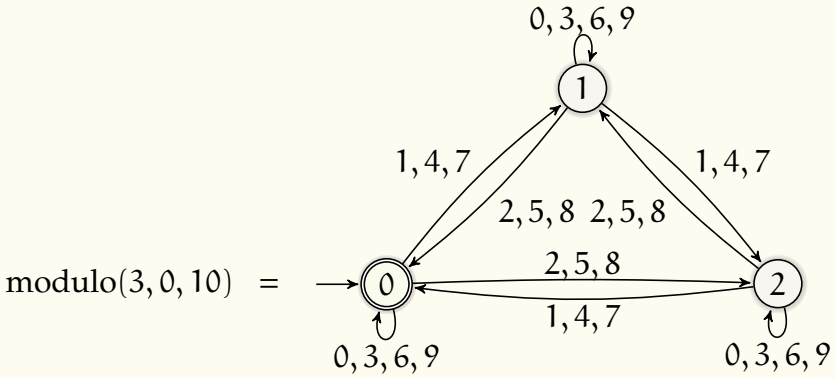
```
D.proj("(]").mini().visu()
```

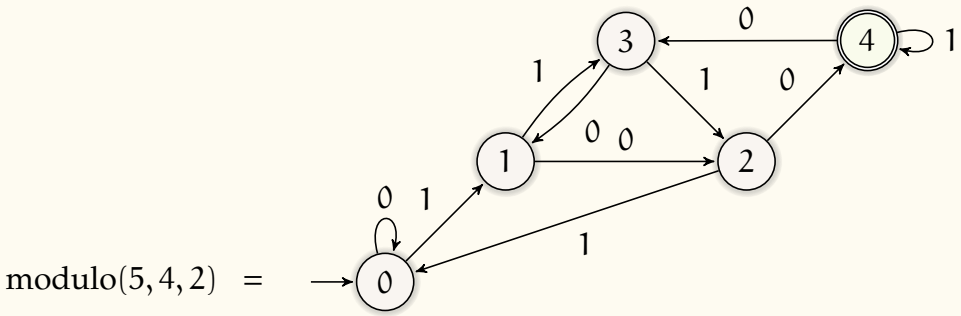
You should obtain a universal automaton.

(7) Modulos

Define and implement automata 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





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:

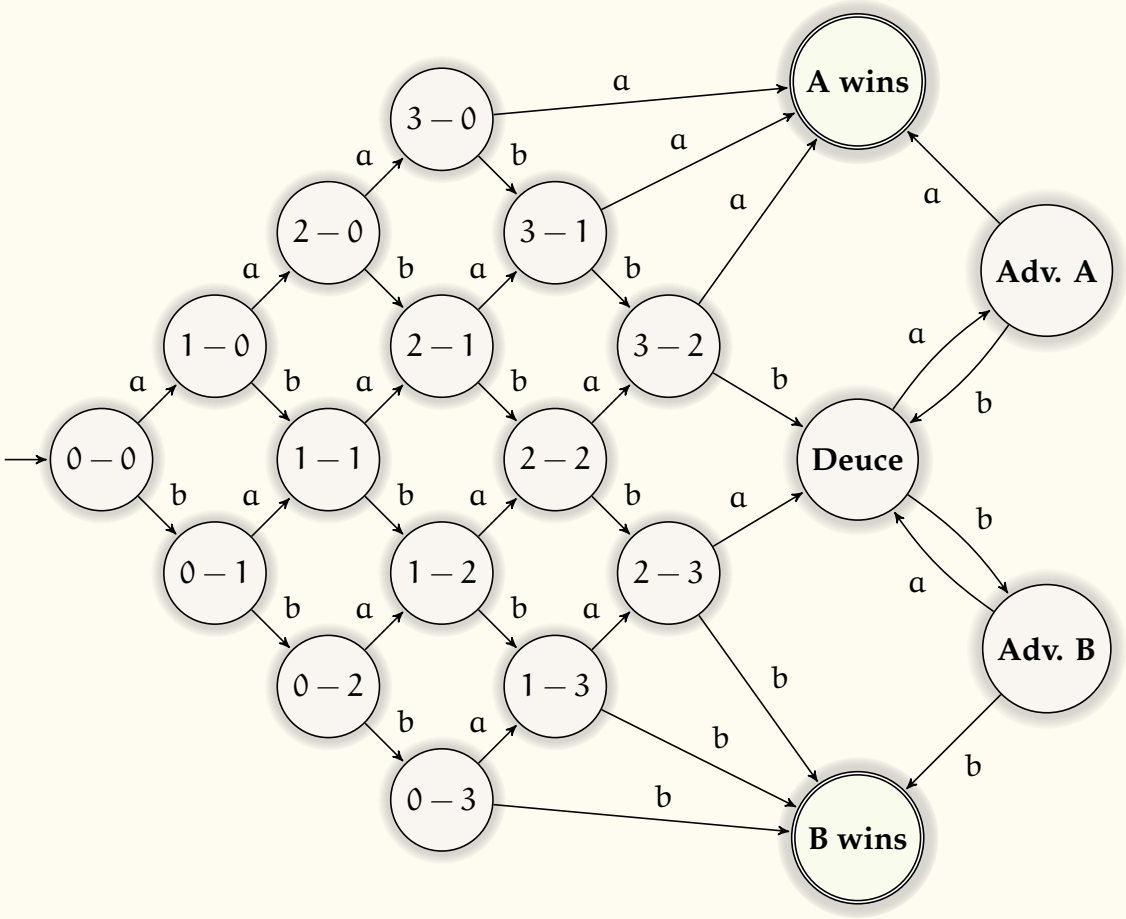
> *A game consists of a sequence of points played with the same player serving. A game is won by the first player to have won at least four points in total and at least two points more than the opponent.*
>
> *[. . .]*
>
> *If at least three points have been scored by each player, making the player's 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.

20 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. 27.2.7[p161]: "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.

21 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 ?

22 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[p116]:

- ◇ 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[p117]:

- ◇ 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 P_0 has just accessed its critical section, then the next one to access its critical section must be P_1, and vice-versa.

Part IV

OLD Lecture Notes: Formal Verification

23 Meta-information about the course **123**
23.1 Note on the notes: . . . 123
23.2 Course prerequisites and student assessment . . . 123

24 Introduction: What is Formal Verification? **124**
24.1 Problem: Disaster stories . . . 124
24.2 Solution: Verification . . . 124
24.3 A Brief History of Program Proof . . . 125
24.4 Our focus in this course: Model Checking . . . 126
24.5 Provisional course plan . . . 126

25 State Systems and Modelisation **127**
25.1 Brief Reminders About Nondeterministic Finite State Automata . . . 127
25.2 On machine: `lecture_automata_products.py` . . . 128
25.3 Modelling through automata: generalities . . . 128
25.4 Examples of Isolated Systems . . . 128
25.4.1 Digital Clock . . . 129
25.4.2 Digicode, pass 123 . . . 129
25.4.3 LIFO (Stack) and FIFO (Queue) of size 2 . . . 131
25.4.4 Incrementable Integer Variable . . . 132

26 Incrementable Unsigned Integer Variable with Overflow **133**

27 Set Variable **134**
27.0.1 FIFO / LIFO($\mathbf{n}, \mathbf{m}$) . . . 135
27.0.2 The Wolf, the Goat, and the Cabbage (WGC) . . . 135
27.0.3 WGC, states as functions rather than "left-bank" sets . . . 138
27.0.4 Indiana Jones and the Temple of Verification . . . 138
27.1 A Taxonomy of Automata Products . . . 140
27.1.1 Fully Synchronised Product $\otimes$ . . . 141
27.1.1.1 Fully Synchronised Product $\otimes$ for $\cap$ . . . 141
27.1.1.2 Fully Synchronised Product $\oplus$ for $\cup$ . . . 142
27.1.2 Fully Unsynchronised Product $\|$: the Shuffle . . . 142
27.1.3 Vector-Synchronised Product . . . 143
27.1.3.1 A Fully General Product . . . 145
27.1.3.2 Easy to Understand, Cumbersome to Use . . . 145
27.1.4 Named Synchronised Product . . . 146
27.1.5 Automaton Restriction . . . 147

27.2 Example Systems, Now With Some Products . . . 148
27.2.1 WGC, Now With Map-Synchronised Product . . . 148
27.2.2 Indiana Jones, now With Map-Synchronised Product . . . 151
27.2.2.1 The Solution, Concisely . . . 151
27.2.2.2 How Was I Supposed to Guess How to Handle Time? . . . 151
27.2.3 Exercise with Solution: The Bridge on the River Kwaï (FR) . . . 152
27.2.3.1 Problem Statement . . . 152
27.2.3.2 Solution . . . 153
27.2.4 Exercise with Solution: The Toggle Problems . . . 155
27.2.4.1 Problem Statement . . . 155
27.2.4.2 Solution . . . 156
27.2.5 Exercise with Solution: The Three Islands, the Two Wolves, the Goat, and the Cabbage . . . 157
27.2.5.1 Problem Statement . . . 157
27.2.5.2 Solution . . . 158
27.2.6 Exercise with Solution: Max-Weight River-Crossing Problem . . . 159
27.2.6.1 Problem Statement . . . 159
27.2.6.2 Solution . . . 160
27.2.7 Semaphores: first contact . . . 161
27.2.8 Mutable Boolean variable . . . 163
27.2.9 Sequential Programs . . . 164
27.2.9.1 Infinite Loop: `while True ...` . . . 165
27.2.9.2 `if ... else ...` . . . 166
27.2.9.3 Sequence of instructions . . . 166
27.2.9.4 Null operation: `pass` . . . 166
27.2.9.5 Application to our example . . . 166
27.2.10 Peterson's Algorithm . . . 167

28 Classical and Temporal Logics: (W)S1S, CTL*, CTL, LTL 170
28.1 Traditional Logic on Words: (w)S1S . . . 171
28.1.1 What Does that Word Salad Even Mean? . . . 172
28.1.2 Formal Syntax and Semantics . . . 174
28.1.2.1 An Example . . . 175
28.1.3 Extending the Syntax; Writing Properties . . . 175
28.1.4 The Büchi and Thatcher–Wright Theorems . . . 178
28.1.4.1 For every NFA, there is an equivalent wS1S formula . . . 178
28.1.4.2 For every wS1S formula, there is an equivalent NFA . . . 180
28.1.5 Algorithmic Complexity and Suitability for Verification . . . 180
28.1.5.1 Reminders about Complexity Theory . . . 181
28.1.5.2 What Does it Mean for Our Purposes? . . . 182
28.2 CTL*: Computation Tree and Linear Time Logic (CTL+LTL+···) . . . 183
28.2.1 Kripke Structures and Paths . . . 184

28.2.2 Syntax and Semantics of CTL* . . . 184
28.2.3 A Few Examples, to Help the Semantics go Down . . . 186
28.2.4 LTL: Linear Time Logic . . . 188
28.2.5 CTL: Computation Tree Logic . . . 189
28.2.6 Some Useful Properties and Miscellaneous Examples . . . 190
28.2.6.1 Mutual Exclusion . . . 190
28.2.6.2 Possible Access, Liveness, “infinitely often” . . . 190
28.2.6.3 Requests Get Answers, Eventually . . . 191
28.2.6.4 No Shoes, No Service . . . 191
28.2.6.5 Interdiction . . . 191
28.2.6.6 Necessary Steps . . . 192
28.2.7 Comparing LTL, CTL, and CTL* . . . 192
28.2.7.1 Expressive Powers . . . 193
28.2.7.2 Algorithmic Complexity . . . 193
28.2.8 CTL Model-Checking Algorithm . . . 193
28.3 Exercises on Logics and Automata . . . 193
28.3.1 Exam 2020–2021 . . . 193
28.3.1.1 Problem Statement . . . 193
28.3.1.2 Solution . . . 194

23 Meta-information about the course

23.1 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.

23.2 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 [(x)] 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.

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

24 Introduction: What is Formal Verification?

24.1 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) . . .

24.2 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. ...

24.3 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.

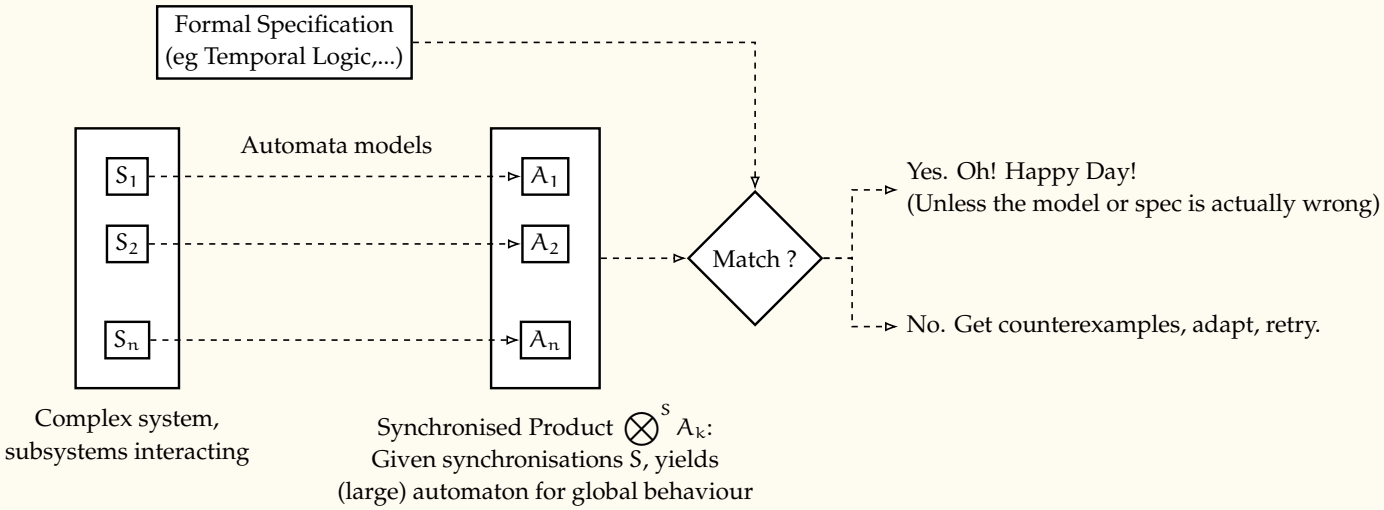
(6) 1980-1990 Nombreux résultats théoriques

(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

24.4 Our focus in this course: Model Checking



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.

24.5 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)

25 State Systems and Modelisation

25.1 Brief Reminders About Nondeterministic Finite State Automata

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

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

Sémantique: langage reconnu par un automate:

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

Déterminisme, non-déterminisme et ε

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

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

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)

25.2 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. 19[p100]: "Basic finite state systems".

25.3 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)

25.4 Examples of Isolated Systems

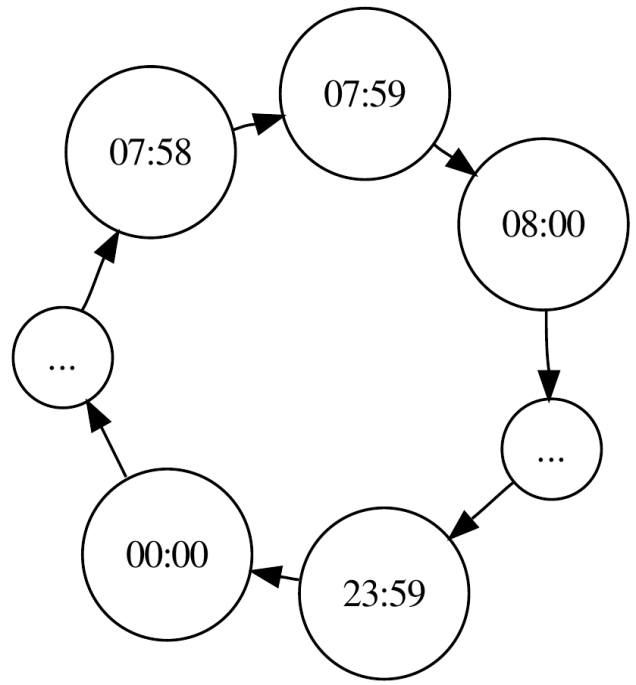
Let us see a few examples of simple systems, and not-so-simple systems that we still tackle by viewing them in their globality, at least for now — we will move towards a "product of

sub-systems" soon enough.

25.4.1 Digital Clock

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

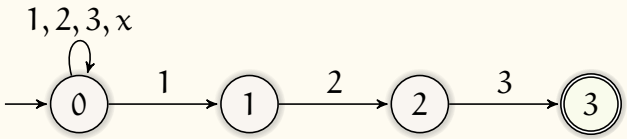
$$24 \times 60 \quad = \quad 1440$$



25.4.2 Digicode, pass 123

You first lab class exercise will be to implement this.

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



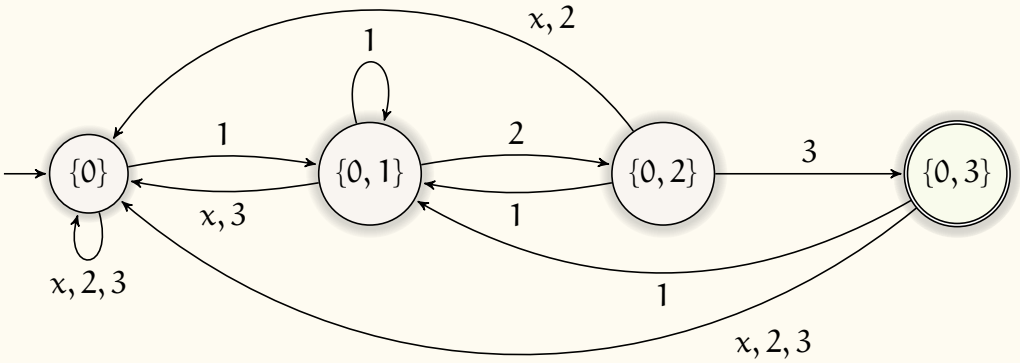
Déterminiser, et écrire une version déterministe directement. Rappels de l'année dernière !

END FIRST LECTURE (2020–2021)

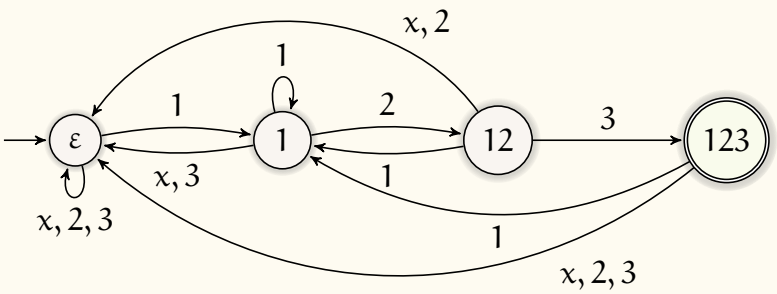
		1	2	3	x
Initial	**0**	0, 1	0	0	0
	1		2		
	2			3	
Final	**3**				

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?".

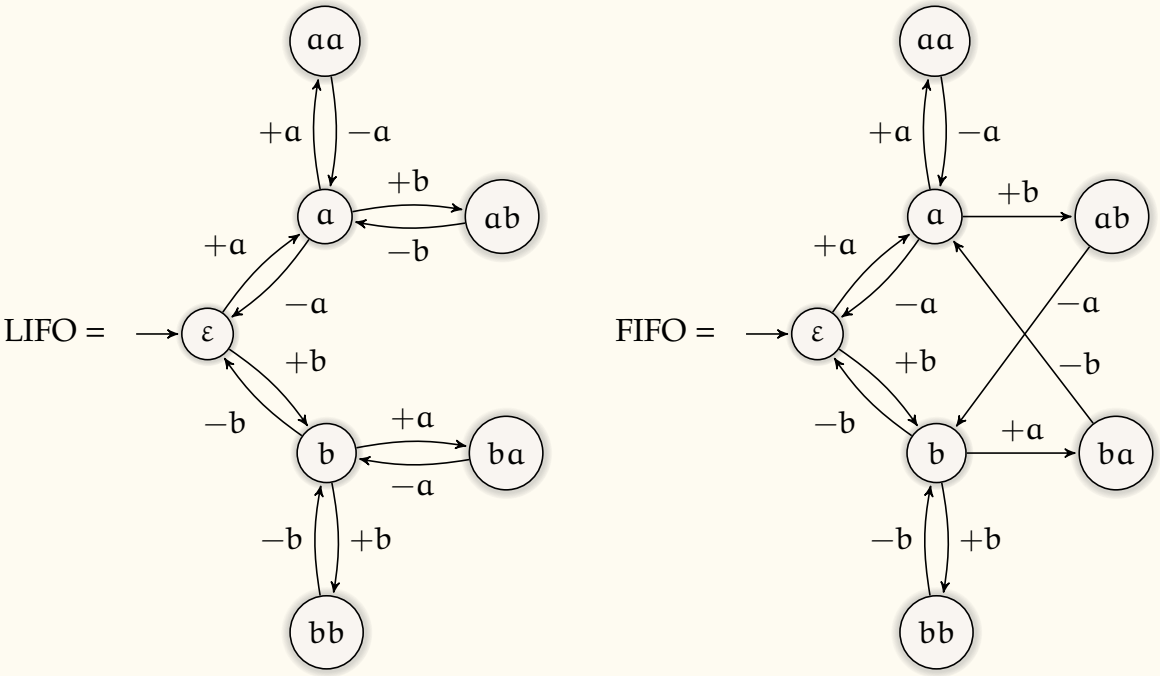


END FIRST LECTURE (2019–2020)

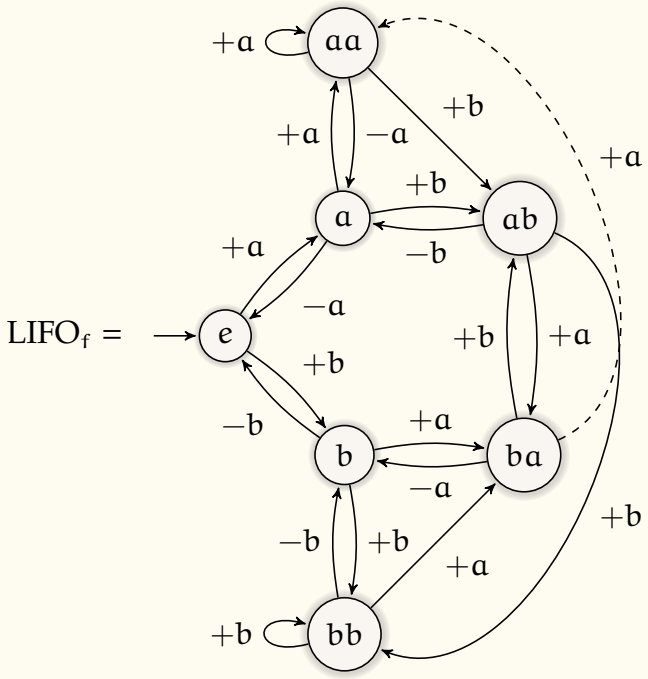
25.4.3 LIFO (Stack) and FIFO (Queue) of size 2

$\Sigma = \{ a, b \}$, 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:



25.4.4 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.

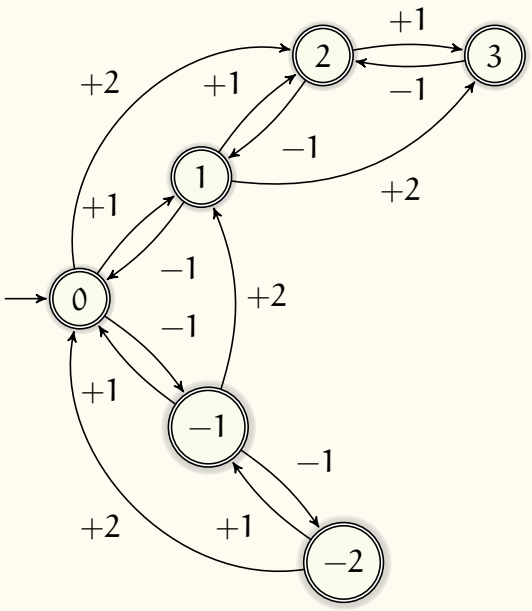
For all $n, m, i \in \mathbb{Z}$ and $X \subseteq \mathbb{Z}$, formally define a NFA $V(n, m, i, X)$ representing an integer variable on the interval $[\![n, m]\!]$, initialised to i, that can be incremented by the quantities in X, and only by those.

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

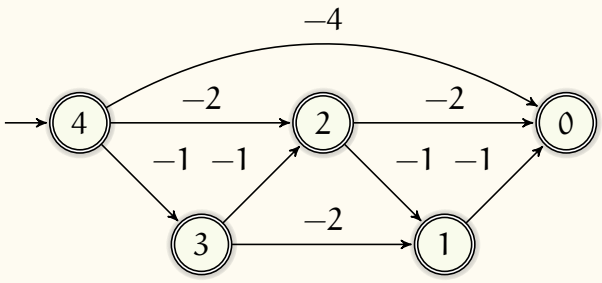
We permit neither overflow nor underflow. We consider all states final.

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})")
```

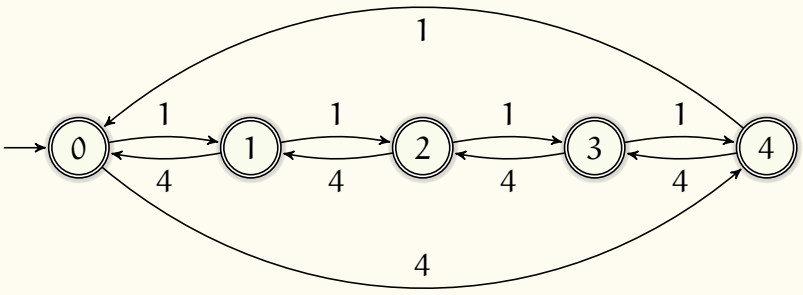
---

26 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

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



The formal definition of V_o is:

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

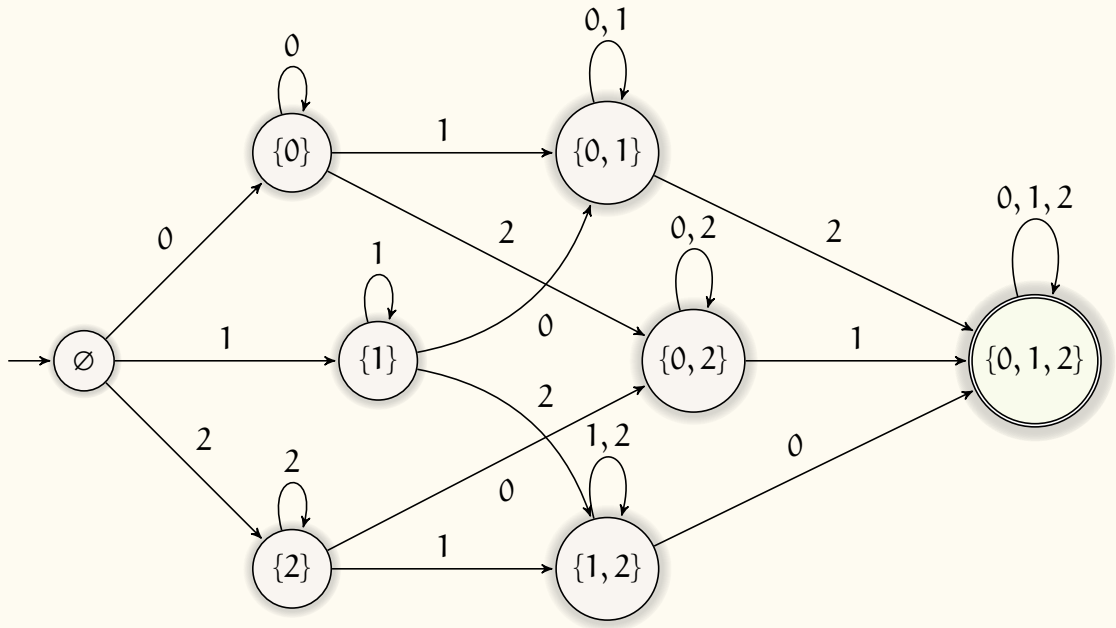
---

27 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:

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

27.0.1 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.

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

Similarly,

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

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:

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

27.0.2 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,

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

and our final states are

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

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

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

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:

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

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.

27.0.3 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

27.0.4 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.

TODO: KID becomes CHILD

> *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:

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

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.

$$Q \subseteq \wp(A) \times [\![0, 15]\!] \tag{27.1}$$

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.

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

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:

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

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

27.1 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

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

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

$$\left(\bigodot_k A_k\right).I = \prod_k A_k.I\,. \tag{27.2}$$

Let us think about what this means. If we have n automata $A_1, \ldots, A_n$, then a state of $B = \bigodot_{k=1}^{n} A_k$ is a tuple $\langle q_1, \ldots, q_n \rangle$ 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 (27.2) 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.

27.1.1 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 $L \cap M = \overline{\overline{L} \cup \overline{M}}$.

27.1.1.1 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`.

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

Although it is rarely used, the same construction, with very minor changes, can be re-purposed 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 $[\![B]\!] = \varnothing$. Then you want $[\![A \oplus B]\!] = [\![A]\!] \cup \varnothing = [\![A]\!]$, 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`.

27.1.2 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 $u_1 v_1 \dots u_n v_n \dots$. All this, and everything in between is possible.

The corresponding language operation is called the **shuffle**, or **interleaving**; we shall denote it by $\parallel$ 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:

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

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

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

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

Demo in `lecture_automata_products.py`.

27.1.3 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

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

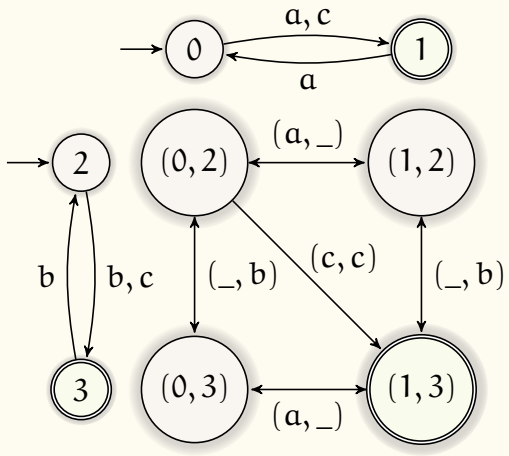
be our **synchronisation set**. Then the **vector-synchronised product of the A_k according to S** is given by

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

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 $S = \{ (a, \_), (\_, b), (c, c) \}$:



27.1.3.1 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) $S = \left\{ \overrightarrow{v} \;\middle|\; \exists k,\ v_k \in \Sigma_k,\ \forall i \neq k,\ v_i = \_ \right\}$, that is to say the set of all vectors of the form $\langle \_, \_, \ldots, a, \ldots, \_ \rangle$: concurrent action on all symbols. Compose with

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

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.

27.1.3.2 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 $\langle W, G, C, F \rangle$, 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.

27.1.4 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

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

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

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

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

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,

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

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

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

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

[(y)] 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.

a transition outside of Farmer.Σ; each component must have its own target space. Formally, we let

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

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

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

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

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

27.1.5 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

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

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

27.2 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

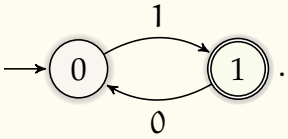
$$\left(\bigotimes^{S} \mathcal{C} \right) . Q \cap B \quad = \quad \varnothing \, .$$

27.2.1 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:

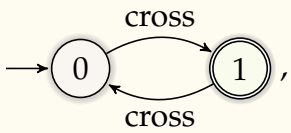
The rule for components changing states is: "the Farmer and at most one other actor on the same bank can jointly change state; nothing else".

Thus we have the synchronisation set:

$$S \;=\; \left\{ \{F : x, a : x\} \;\middle|\; x \in \{0, 1\},\ a \in \mathcal{C} \right\}. \tag{27.3}$$

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. (27.3). The closest we can come in the vector version is to define a set

$$M \;=\; \big\{\, \{F, a\} \;\big|\; a \in \mathcal{C} \,\big\}$$

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,

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

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

$$q^{-1}[S] \;=\; \big\{\, x \;\big|\; q(x) \in S \,\big\}$$

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

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

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

27.2.2 Indiana Jones, now With Map-Synchronised Product

Let us come back to Sec. 27.0.4[p138]: "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.

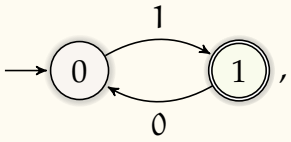
27.2.2.1 *The Solution, Concisely*

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. 25.4.4[p132]: "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

27.2.2.2 *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. 27.0.4[p138]: "Indiana Jones and the Temple of Verification", Eq. (27.1):

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

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.

27.2.3 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.

27.2.3.1 *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 $\bigotimes^S \mathcal{C}$. [(z)]

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.

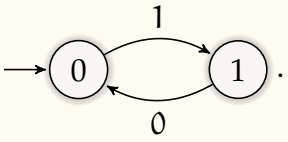
(4) $\mathcal{C}$ change-t-il ? Si oui comment ?

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

(6) Même question pour ℓ.

27.2.3.2 *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).

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

[(z)] 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.

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:

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

(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:

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

(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:

$$\begin{aligned} P \;&=\; V(0, N, 0, \{1, 2, 4\}) \\ \mathcal{C}' \;&=\; \mathcal{C} \cup \{P\} \end{aligned}$$

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:

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

(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)

27.2.4 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[(aa)]. I call them "toggle puzzles".

27.2.4.1 *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.*

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

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

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:

[(aa)] 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.

You find N *switches* $s_1, \ldots, s_N$*, initially all off, and the goal is to turn them all on.*

They are linked by a "flipping function"

$$f \in \{ s_1, \ldots, s_N \} \to \wp(\{ s_1, \ldots, s_N \})$$

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.

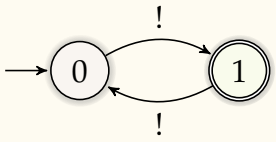
(6) Give, if useful, a filter ℓ upon the states of the synchronised system $\bigotimes^S \mathcal{C}$.

27.2.4.2 *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:

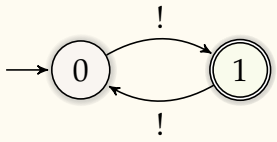
$$S \quad = \quad \big\{ \{ s_1 : !,\ s_3 : ! \}, \{ s_2 : !,\ s_1 : ! \}, \{ s_3 : !,\ s_2 : !,\ s_4 : ! \}, \{ s_4 : !,\ s_1 : ! \} \big\}$$

(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.*

$s_1, \ldots, s_N \quad =$



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

(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.

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

This was an exercise in the 2021 final exam.

27.2.5.1 *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

$$P = \left. \bigotimes^S \mathcal{C} \right|_\ell .$$

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.

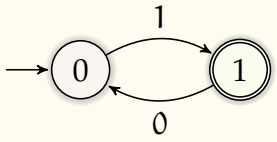
Tip: Think carefully. Actors must only be able to go together to island X if they are *both* on the same island Y. Make sure your model has enough information to enforce that.

(2) Give the set S of synchronisation maps.

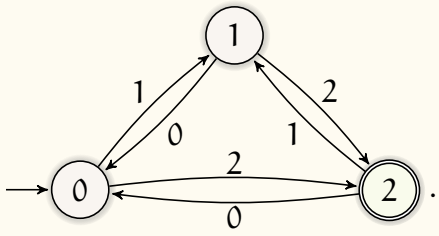
(3) Give, if useful, a filter ℓ upon the states of the synchronised system $\bigotimes^S \mathcal{C}$.

27.2.5.2 *Solution*

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

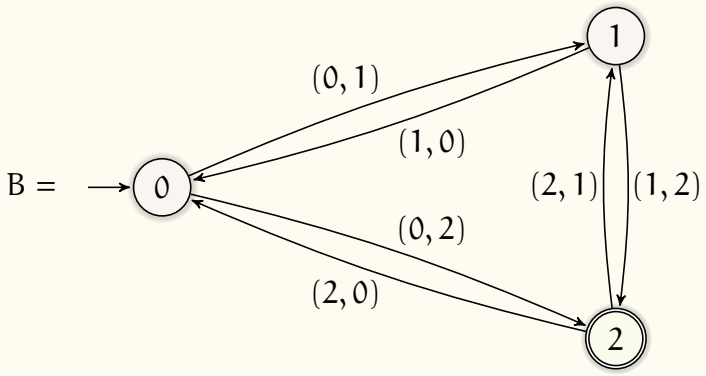


to the obvious three-banks / three-islands counterpart:

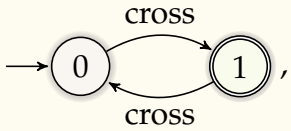


However, as famously remarked by Admiral Ackbar (back when Star Wars was cool), ***this is a trap*!** Why? Because synchronising on 2 would enable an actor on 0 and an actor on 1 to both teleport to 2. This is illegal! They can only move together if they are on the same island, as there is only one boat.

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

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

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:

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

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

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

and all islands must be licit:

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

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

This was an exercise in the 2021 final exam.

27.2.6.1 *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

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

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 $\bigotimes^{S} \mathcal{C}$.

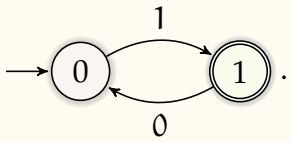
27.2.6.2 *Solution*

(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:

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

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

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

27.2.7 Semaphores: first contact

After this lecture, you can do Exercise (16)[p116]

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.*
>
> *[...]*

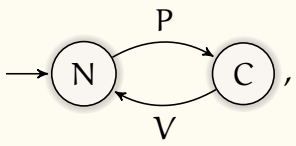
> *To avoid starvation (*famine*), a semaphore has an associated queue (*file*) of processes (usually with FIFO semantics). If a process performs a P operation on a semaphore that has the value zero, the process is added to the semaphore's queue and its execution is suspended. When another process increments 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
```

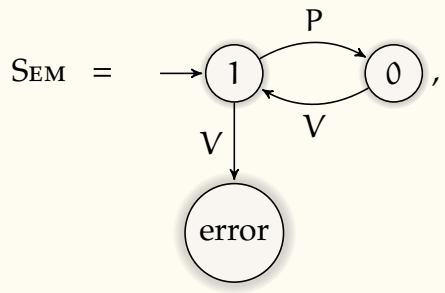
Let us represent this as an automaton, with N for "noncritical section" and C for "critical section" for each of the two processes, and 0, 1 for the semaphore. We have a total of $2^3 = 8$ possible states. Are all of them desirable? Probably not; we should avoid having the two processes in critical section 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

$P_0, P_1 =$ 

,

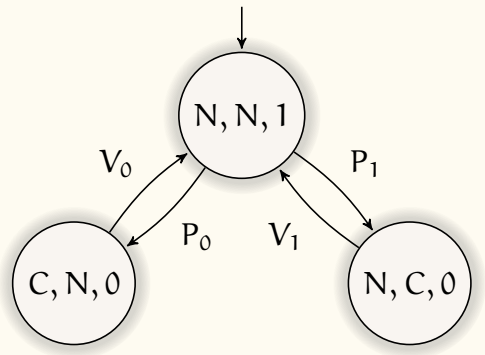
for the processes P_0 and P_1, and an instance of



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

$$S = \left\{ \{ \text{Sem} : x, P_i : x \} \;\middle|\; x \in \{P, V\},\ i \in \{0, 1\} \right\},$$

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...

27.2.8 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.

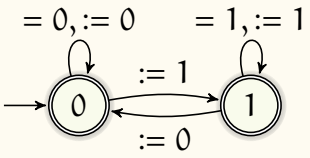
Let's deal with Boolean variables, and this time, let us model both *read* and *write* interactions with other components. The same approach can of course be applied to the other types of variables seen before to add support for read interactions.

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")
```

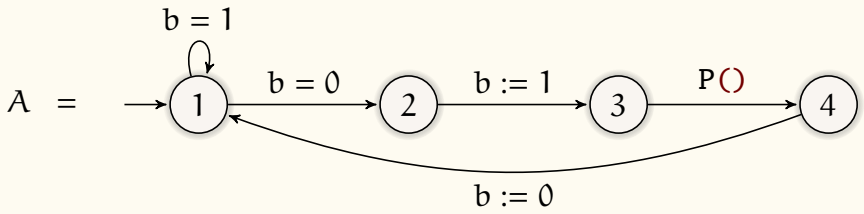
27.2.9 Sequential Programs

Consider the reactive program

```
while True:      # 11
  if not b:
    b := True    # 12
    P()          # 13
    b := False   # 14
```

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.

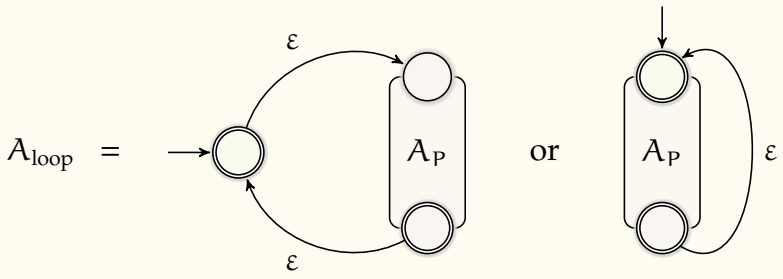
For now, this control flow automaton is obtained through intuition. Is there a systematic way to construct it? Let us examine the principal syntactic constructions and see how we can, inductively, "compile" an Abstract Syntax Tree (AST) into a control-flow automaton.

27.2.9.1 *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:



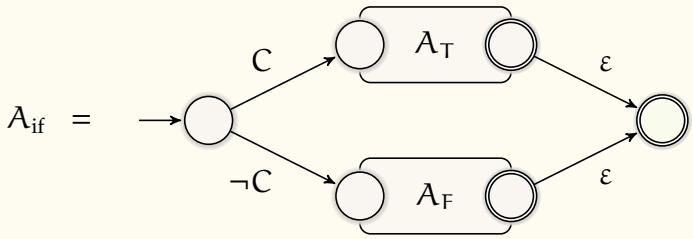
Note that the colour of the accepting state of the sub-automaton is slightly different, to indicate that it is not actually a final state of A_{loop}, just the "output point" of the subautomaton.

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

Now let us do the same with other patterns.

27.2.9.2 *if ... else ...*

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

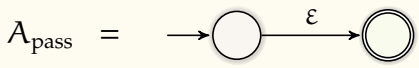


27.2.9.3 *Sequence of instructions*

```
I1
I2
```

27.2.9.4 *Null operation:* ***pass***

```
pass
```

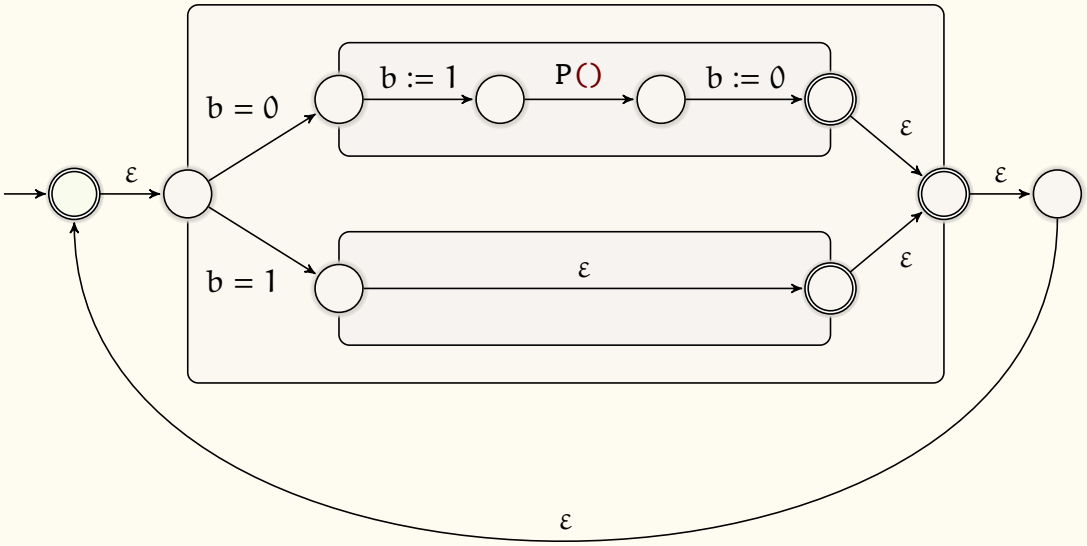


27.2.9.5 *Application to our example*

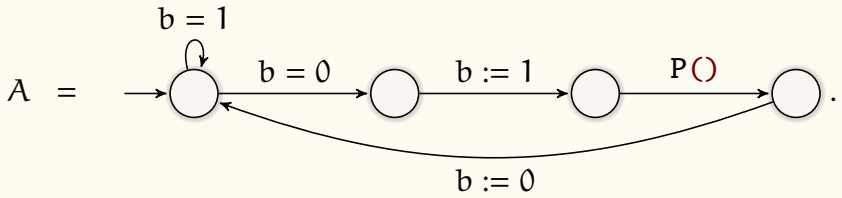
While there are of course other patterns, we now have enough to combine all those translation rules and apply them to our original reactive program example, which we rewrite a little bit:

```
while True:      # 11
  if not b:
    b := True    # 12
    P()          # 13
    b := False   # 14
  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.

27.2.10 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:

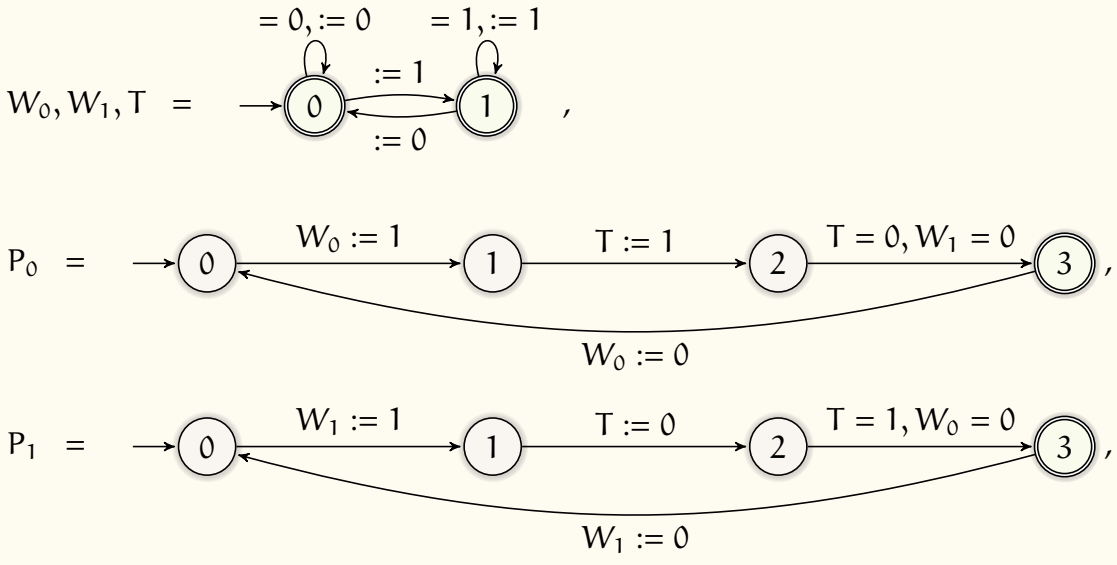
> *Peterson's algorithm (or Peterson's solution) is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication, provided changes to binary variables (bits) propagate immediately 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.[ab]

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
    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):



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

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.

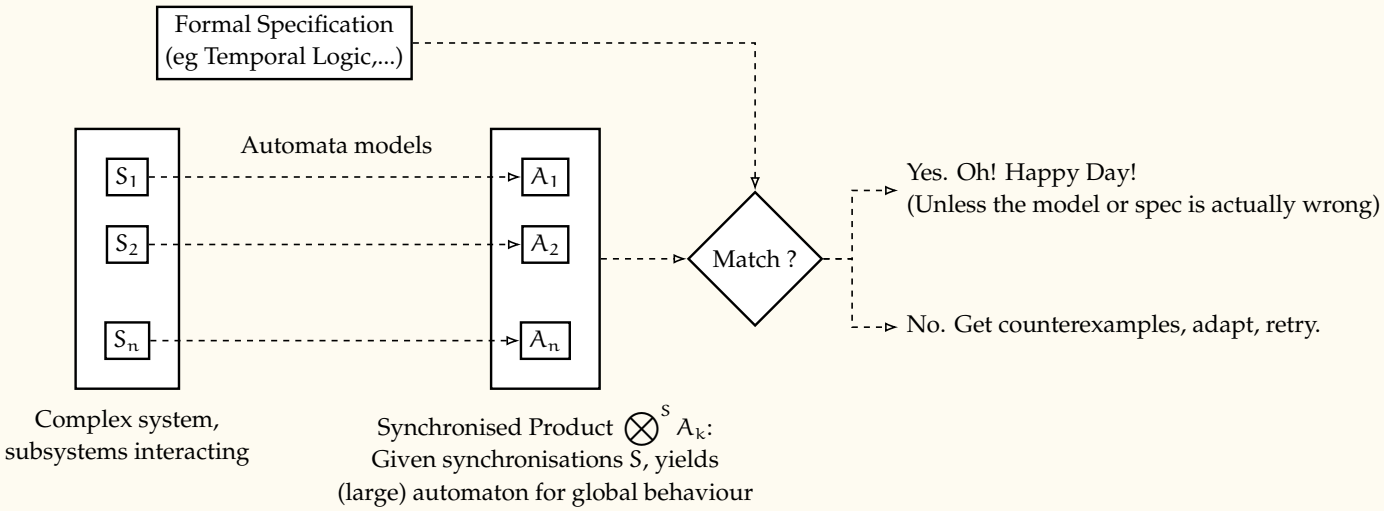
This is all very nice. So far we have managed to obtain a rigorous and systematic method to formally model complex concurrent systems.

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.

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



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.[ac] 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

$$[\![A_S]\!] \subseteq [\![A_P]\!] \,?$$

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.

28.1 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,
- ◇ DSPACE(O(1)) & 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,

[ac] 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.

- ◇ 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.

28.1.1 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.

28.1.2 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 $\mathbb{x} = x, x_1, x_2, \ldots, y, \ldots$ be our first-order variables, and $\mathbb{X} = X, X_1, X_2, \ldots$ be our second-order variables, which here just means they are set variables. Of course we assume $\mathbb{x} \cap \mathbb{X} = \varnothing$. We have the syntax:

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

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:

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

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

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

be the **language described by φ**.

END FIFTH LECTURE (2021–2022)

28.1.2.1 *An Example*

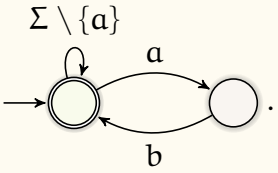
Let us consider a very simple property, of the form "every request a is immediately followed by a response b". This is coded by the following formula:

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

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.

28.1.3 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)

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

(8) Set emptiness:

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

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

(9) Set inclusion:

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

(10) Set equality:

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

(11) Set intersection and union (binary):

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

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

(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:

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

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

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

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

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

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 $\mathrm{wS1S}_0$, 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:

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

(15) Set X is closed above:

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

(16) Inequalities:

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

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

(17) Ranges:

$$x \in [\![i, j]\!] \quad \equiv \quad i \leqslant x \leqslant j \quad \equiv \quad i \leqslant x \ \wedge \ x \leqslant j$$

28.1.4 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.*

28.1.4.1 *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 $A = \langle \Sigma, Q, I, F, \Delta \rangle$, there is an equivalent formula φ.

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

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

We can obtain $\bot$ 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:

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

We build φ following the idea that we encode a successful run of 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:

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

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 $\leftrightarrow$ 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.

28.1.4.2 *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.

28.1.5 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

$$[\![\varphi]\!] = \varnothing \quad \Leftrightarrow \quad [\![\mathcal{A}_\varphi]\!] = \varnothing \,.$$

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 $\mathcal{A}_\varphi$, 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 $\mathcal{A}_\varphi$ 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:

$$[\![\mathcal{A}_S]\!] \subseteq [\![\mathcal{A}_\varphi]\!] \,.$$

Concretely this computation will be effected as

$$[\![\mathcal{A}_S]\!] \cap \overline{[\![\mathcal{A}_\varphi]\!]} = \varnothing \,.$$

which means we need an automaton to complement and involve in a product. The size of $\mathcal{A}_\varphi$ needs to be reasonable for this programme to be applicable in practice.

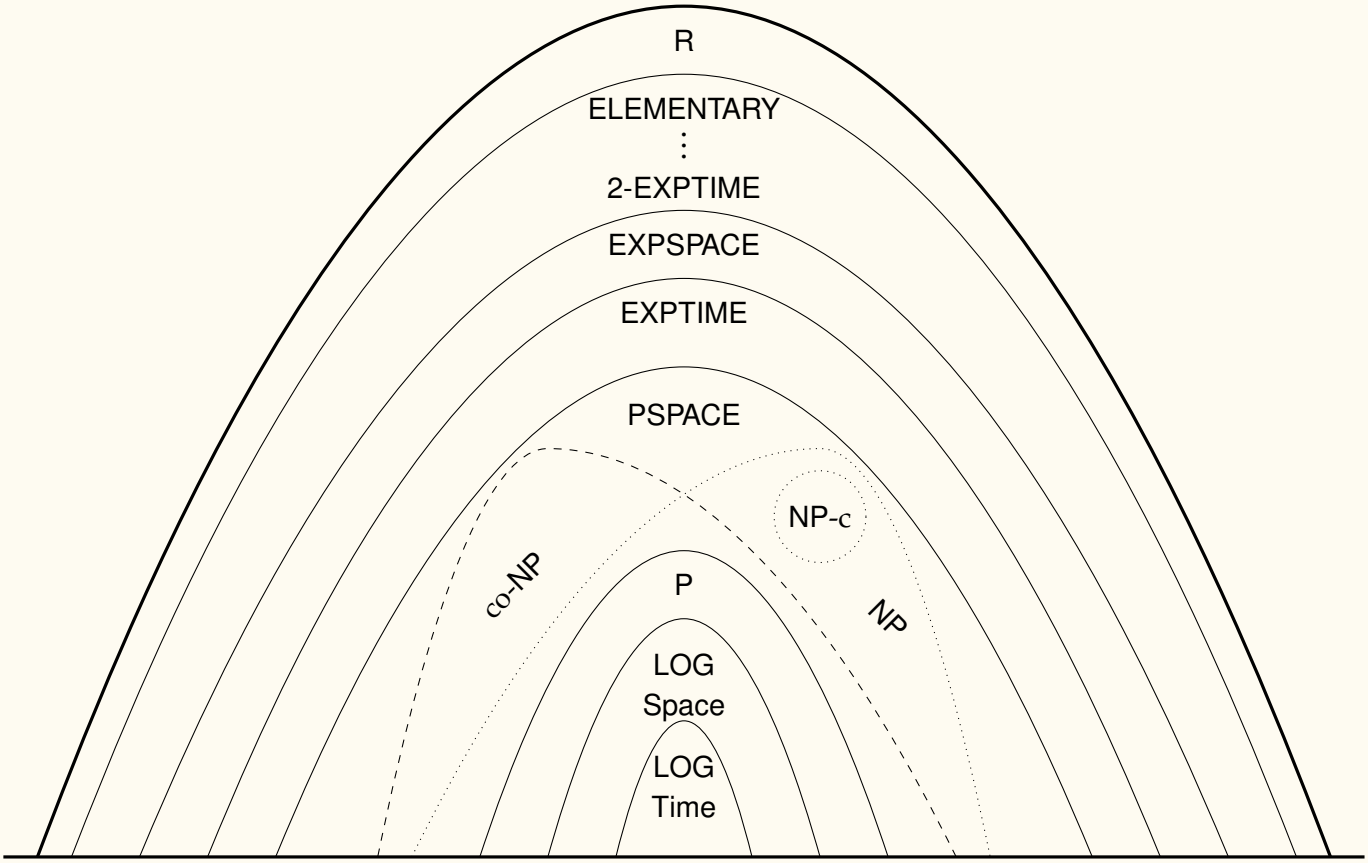


Figure 1: Computational Complexity Classes.

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.

28.1.5.1 Reminders about Complexity Theory

Figure 1 [p181] [(ad)] 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,

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DSPACE}\left(f(n)^2\right) . \qquad \text{(Savitch 1970)}$$

[(ad)] Slight modifications on a figure by Sebastian Sardina.

The main classes are as follows:

$$\mathsf{P} = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}(n^k) = \mathsf{DTIME}(n^{O(1)})$$

$$\mathsf{NP} = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(n^k) = \mathsf{NTIME}(n^{O(1)})$$

$$\mathsf{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{DSPACE}(n^k) = \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}(n^k)$$

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.

$$\mathsf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}\left(2^{n^k}\right)$$

$$\mathsf{NEXPTIME} = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(2^{n^k})$$

$$\mathsf{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{DSPACE}\left(2^{n^k}\right) = \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}\left(2^{n^k}\right)$$

$$\text{2-}\mathsf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}\left(2^{2^{n^k}}\right)$$

This generalises to higher and higher towers of exponentials:

$$\mathrm{N}\text{-}\mathsf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}\left(\underbrace{2^{2^{\cdots^{2^{n^k}}}}}_{\text{Tower of size N}}\right),$$

And then we have

$$\begin{aligned}\mathsf{ELEMENTARY} &= \bigcup_{k\in\mathbb{N}} k\text{-}\mathsf{EXPTIME} \\ &= \mathsf{DTIME}(2^n) \cup \mathsf{DTIME}\left(2^{2^n}\right) \cup \mathsf{DTIME}\left(2^{2^{2^n}}\right) \cup \cdots,\end{aligned}$$

the class of problems decidable in time that is bounded by tower of exponentials of arbitrary height. This is our stop.

28.1.5.2 *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.

28.2 CTL*: Computation Tree and Linear Time Logic (CTL+LTL+···)

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 $\mathrm{FO}^2(*)$, 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*.

28.2.1 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

$$\Pi(q) \;=\; \left\{ \pi \in Q^\omega \;\middle|\; \pi[0] = q \;\wedge\; \forall k \in \mathbb{N},\, \pi[k] \to \pi[k+1] \in \Delta \right\}$$

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)

28.2.2 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:

$$\varphi \in \text{CTL* (state)} \quad ::= \quad p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists\psi \qquad\qquad p \in \mathbb{A}$$

$$\psi \in \text{CTL* (path)} \quad ::= \quad \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \circ\psi \mid \psi \,\mathbf{U}\, \psi$$

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:

$$\begin{array}{lcl} q \models p & \Leftrightarrow & p \in \ell(q) \\ q \models \neg\varphi & \Leftrightarrow & q \not\models \varphi \\ q \models \varphi \wedge \varphi' & \Leftrightarrow & q \models \varphi \ \wedge \ q \models \varphi' \\ q \models \exists\psi & \Leftrightarrow & \exists \pi \in \Pi(q) : \ \pi \models \psi \end{array}$$

A path $\pi \in Q^\omega$ satisfies a path formula ψ under the following conditions:

$$\begin{array}{lcl} \pi \models \varphi & \Leftrightarrow & \pi[0] \models \varphi \\ \pi \models \neg\psi & \Leftrightarrow & \pi \not\models \psi \\ \pi \models \psi \wedge \psi' & \Leftrightarrow & \pi \models \psi \ \wedge \ \pi \models \psi' \\ \pi \models \circ\psi & \Leftrightarrow & \pi[1:] \models \psi \\ \pi \models \psi \,\mathbf{U}\, \psi' & \Leftrightarrow & \exists k \in \mathbb{N} : (\forall i \in [\![0, k[\![, \ \pi[i:] \models \psi) \ \wedge \ \pi[k:] \models \psi' \end{array}$$

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 $\top, \bot, \vee, \Rightarrow, \iff$, 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 [ae]" (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.

Since $\Box$ and $\Diamond$ are so useful, let us state their semantics explicitly, while noting that it is entirely derived from that of **U**:

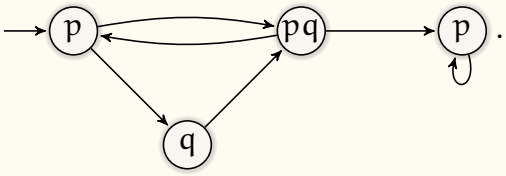
$$\begin{array}{lcl} \pi \models \Box\psi & \Leftrightarrow & \forall k \in \mathbb{N} : \ \pi[k:] \models \psi \\ \pi \models \Diamond\psi & \Leftrightarrow & \exists k \in \mathbb{N} : \ \pi[k:] \models \psi \end{array}$$

[ae] *fatalement* in French

28.2.3 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

```
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,

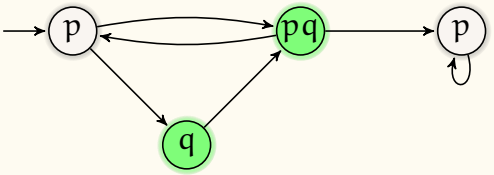
```
checkvisu( kat,labels,
  (EU, p, (AND, (NOT, p), (AU, (NOT, p), q))),
  visu=("simple","detailed") )
```

offers two different visualisations of the model-checking of our system `kat`, with labelling dictionary `labels`, for the formula

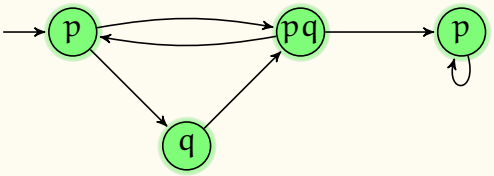
$$\exists\Big(p \,\mathbf{U}\, \big(\neg p \wedge \forall(\neg p \,\mathbf{U}\, q)\big)\Big) \,.$$

Demo in `test.py` for complicated (CTL) examples.

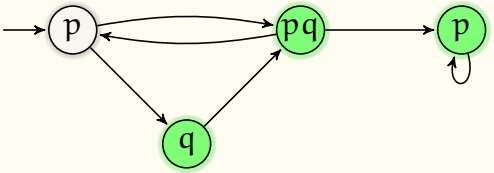
(1) q:



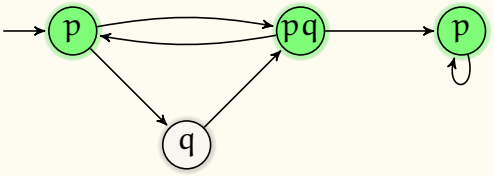
(2) $\exists\circ p$:



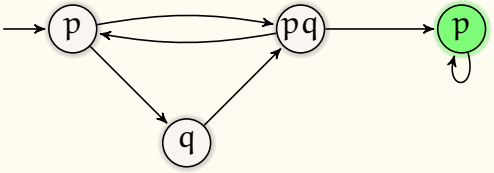
(3) $\forall\circ p$:



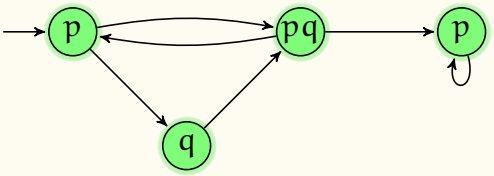
(4) $\exists\Box p$:



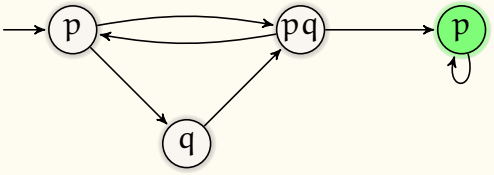
(5) $\forall\Box p$:



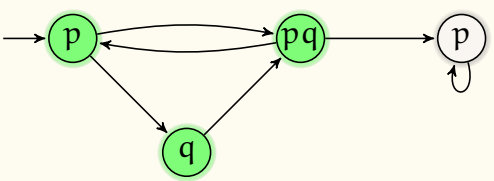
(6) $\exists\Diamond\exists\Box p$
$\exists\Diamond\forall\Box p$
$\forall\Diamond\exists\Box p$



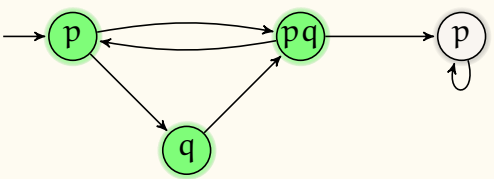
(7) $\forall\Diamond\forall\Box p$:



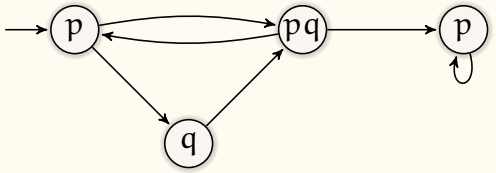
(8) $\forall(p\ \mathbf{U}\ q)$
$\exists(p\ \mathbf{U}\ q)$



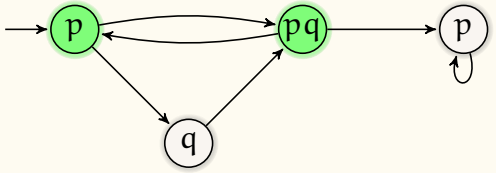
(9) $\exists\Big(p\ \mathbf{U}\ \big(\neg p \wedge \forall(\neg p\ \mathbf{U}\ q)\big)\Big)$:



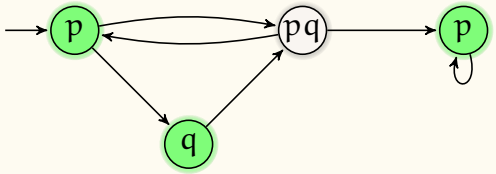
(10) $\forall\circ\forall\circ q$
$\forall\circ\exists\circ q$



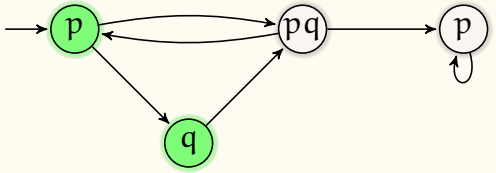
(11) $\exists\circ\forall\circ q$
$\exists\circ\exists\circ q$



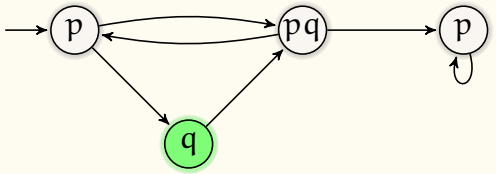
(12) $\forall\circ\circ p$ (that one is LTL):



$\forall\circ\Diamond(p \wedge q)$ (LTL)
(13) $\forall\Diamond\circ(p \wedge q)$ (LTL)
$\forall\circ\forall\Diamond(p \wedge q)$



(14) $\forall\Diamond\forall\circ(p \wedge q)$:



28.2.4 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*:

$$\varphi ::= \forall\psi \qquad \psi ::= p \mid \neg\psi \mid \psi \wedge \psi \mid \circ\psi \mid \psi \,\mathbf{U}\, \psi \qquad p \in \mathbb{A}$$

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 $\forall\Diamond p$.

28.2.5 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*:

$$\varphi \in \text{CTL (state)} \quad ::= \quad p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists\psi \qquad\qquad p \in \mathbb{A}$$

$$\psi \in \text{CTL (path)} \quad ::= \quad \neg\psi \mid \circ\varphi \mid \varphi \mathbf{U} \varphi$$

Now let us see a more direct syntax:

$$\varphi \in \text{CTL} \quad ::= \quad p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists\circ\varphi \mid \exists[\varphi \mathbf{U} \varphi] \mid \forall[\varphi \mathbf{U} \varphi]$$

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:

$$\begin{aligned}
\exists\Diamond\varphi &\equiv \exists[\top \mathbf{U} \varphi] && \text{"potentially holds"} \\
\forall\Diamond\varphi &\equiv \forall[\top \mathbf{U} \varphi] && \text{"is inevitable"} \\
\exists\Box\varphi &\equiv \neg\forall\Diamond\neg\varphi && \text{"potentially always"} \\
\forall\Box\varphi &\equiv \neg\exists\Diamond\neg\varphi && \text{"invariantly"} \\
\forall\circ\varphi &\equiv \neg\exists\circ\neg\varphi && \text{"for all paths next"}
\end{aligned}$$

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

$$\begin{aligned}
\exists\neg\circ p &\equiv \exists\circ\neg p \\
\exists(\circ\varphi_1 \wedge \circ\varphi_2) &\equiv \exists\circ(\varphi_1 \wedge \varphi_2) \\
&\dots
\end{aligned}$$

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.

28.2.6 Some Useful Properties and Miscellaneous Examples

28.2.6.1 *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.

28.2.6.2 *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:

$$\forall\Big(\big(\Box\Diamond W_1 \Rightarrow \Box\Diamond C_1\big) \ \wedge\ \big(\Box\Diamond W_2 \Rightarrow \Box\Diamond C_2\big)\Big) \qquad \text{in LTL}$$

28.2.6.3 *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

$$\forall\left(\Box\left[R \Rightarrow \bigvee_{0\leqslant i\leqslant k} \circ^i A\right]\right) \qquad \text{in LTL,}$$

where $\circ^i$ means, of course, $\underbrace{\circ\circ\cdots\circ}_{i}$. Alternatively,

$$\forall\Box\Big(R \Rightarrow \bigvee_{0\leqslant i\leqslant k} (\forall\circ)^i A\Big) \qquad \text{in CTL.}$$

28.2.6.4 *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"...

28.2.6.5 *Interdiction*

The property "after an interdiction is issued (I), the event (E) does never happen again" is written

$$\forall\big(\Box(I \implies \Box\neg E)\big) \qquad \text{in LTL}$$

$$\forall\Box\big(I \implies \forall\Box\neg E\big) \qquad \text{in CTL.}$$

28.2.6.6 *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

$$\forall\big(\Diamond \mathrm{B} \implies \mathrm{P}\,\mathbf{U}\,\mathrm{B}\big) \qquad \text{in LTL.} \tag{28.1}$$

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

$$\forall\Diamond \mathrm{B} \implies \forall[\mathrm{P}\,\mathbf{U}\,\mathrm{B}]$$

does not have the same meaning as (28.1). 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

$$\forall\Box\big(\Diamond W \implies \mathrm{S}\big) \qquad \text{in LTL.}$$

There again, that property is not expressible in CTL. Consider for instance

$$\forall\Box\big(\forall\Diamond W \implies \mathrm{S}\big)\,.$$

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.

28.2.7 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*.

28.2.7.1 *Expressive Powers*

28.2.7.2 *Algorithmic Complexity*

28.2.8 CTL Model-Checking Algorithm

28.3 Exercises on Logics and Automata

28.3.1 Exam 2020–2021

28.3.1.1 *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:

$$m : \left| \begin{array}{ccc} \Sigma^* & \longrightarrow & \wp(\Sigma)^\omega \\ a_1, \ldots, a_n & \longmapsto & \{a_1\} \ldots \{a_n\}\varnothing^\omega \end{array} \right. ,$$

and we define

$$[\![\psi]\!] \;=\; \{ w \in \Sigma^* \mid m(w) \models \psi \} .$$

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:

$$\psi_1 \;=\; \Box(a \Rightarrow \circ\Box\neg a)$$

(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

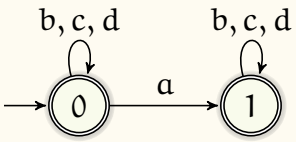
$$\psi_3 \;=\; \Box\big(a \Rightarrow \Diamond(b \wedge \circ b \wedge \Diamond c)\big) .$$

28.3.1.2 *Solution*

(1)

$$\square(a \Rightarrow \circ\square\neg a)$$

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 $\square$, 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)

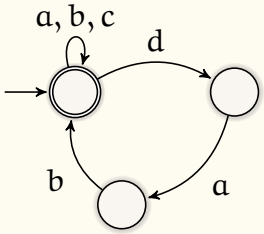
$$\forall x,\ a(x) \Rightarrow (\forall y,\ y > x \Rightarrow \neg a(y))$$

(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.



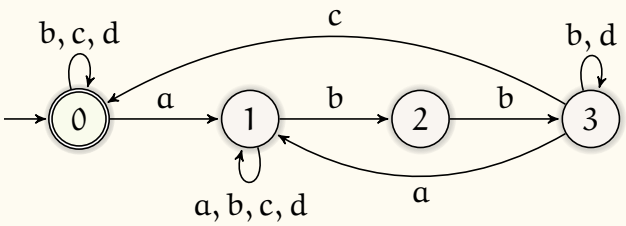
c. Give an LTL path formula ψ_2 equivalent to this property.

$$\square\big(d \Rightarrow (\circ a \wedge \circ\circ b)\big)\ ,$$

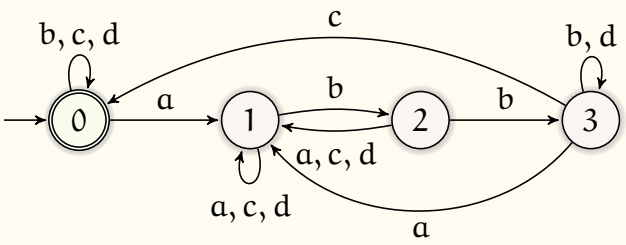
(4)

$$\square\big(a \Rightarrow \diamond(b \wedge \circ b \wedge \diamond c)\big)$$

Corresponds to the NFA



or DFA



I Generalities 7

1 Meta-information about the course 8
1.1 Note on the notes: . . . 8
1.2 Course prerequisites and student assessment . . . 8

2 Introduction: What is Formal Verification? 9
2.1 Problem: Disaster stories . . . 9
2.2 Solution: Verification . . . 9
2.3 A Brief History of Program Proof . . . 10
2.4 Our focus in this course: Model Checking with TLA specifications . . . 11
2.5 Provisional course plan . . . 11

3 State Systems and Modelisation 11
3.1 Brief Reminders About Nondeterministic Finite State Automata . . . 11
3.2 Modelling through automata: generalities . . . 13

4 The TLA Toolbox 13
4.1 Linux or Windows? . . . 13
4.2 Install or download the TLA toolbox . . . 13
4.3 Install Graphviz/dot . . . 13
4.4 Dark Theme issues . . . 14
4.5 Display Scaling issues . . . 14
4.6 Annoying modification history comment . . . 14
4.7 Annoying model-checking history . . . 15

II Solving Problems via TLA$^+$ Models 16

5 The Wolf, the Goat, and the Cabbage (WGC) 19
5.1 Basic definitions / vocabulary . . . 21
5.2 `VARIABLES`: What are the system's states? . . . 22
5.3 `Init`: Where do we start? . . . 23
5.4 `Next`: How does the system move? . . . 23
5.4.1 A trivial `Next`, and parameters for the model . . . 24
5.4.2 Now for the real `Next` . . . 26
5.5 Invariants: types and other properties . . . 29
5.6 Reachability as `Target` invariant . . . 31
5.6.1 Traps of reachability as "error" traces . . . 32
5.6.2 Exploring traces . . . 33
5.6.3 Shortest trace? . . . 34

6 LAB CLASS: WGC 35

7 Indiana Jones and the Temple of Verification **36**
7.1 **EXTENDS**: using / importing other modules . . . 38
7.2 Functions and records . . . 39
7.3 Debugging with `Print` . . . 39
7.4 Tightening the problem's vocabulary . . . 40
7.5 The CHOOSE (∋) "quantifier", weird and dangerous? . . . 40
7.5.1 Beware of impossible choices! . . . 40
7.5.2 ∋ vs. ∃: understanding non-determinism . . . 41
7.5.3 Aside on notation, history, and hidden power . . . 42
7.6 No overloading / polymorphism . . . 43
7.7 Identifying inputs / parameters of the problem . . . 43
7.8 **ASSUME**: check your inputs . . . 44

8 LAB CLASS: Indy **45**

9 Semaphores, Peterson's Algorithm, and Temporal Logics **48**
9.1 Semaphores . . . 49
9.2 Semaphores as a PlusCal algorithmic specification . . . 51
9.3 PlusCal in the toolbox: understanding the translation . . . 52
9.3.1 Program Control . . . 54
9.3.2 The `Spec` formula was here all along . . . 54
9.4 Invariants were Temporal (Safety) properties all along! . . . 55
9.5 Liveness Properties: poor starving semaphores! . . . 57
9.6 You can't avoid thinking about the underlying system . . . 58
9.7 Peterson's algorithm . . . 59
9.8 Safety: no problem . . . 60
9.9 What you want, you get! . . . 61
9.10 . . . but who cares what you want? . . . 62
9.11 Fat or starving, at the Scheduler's mercy . . . 63
9.12 The Temporal Logics: CTL*, CTL, & LTL . . . 63

10 The Three Islands, the Two Wolves, the Goat, and the Cabbage **65**
10.1 Vocabulary and Inputs . . . 67
10.1.1 Abstraction is cutting away irrelevant details: do more for less . . . 67
10.1.2 A good abstraction often has fewer primitive concepts . . . 68
10.2 **VARIABLES**: `[A → Locs]` vs. `[Locs → ℘ A]`, for you and TLC . . . 69
10.3 Fifty shades of `Init`: writing and altering functions . . . 71
10.3.1 TLC doesn't know what you know . . . 71
10.3.2 No convenient ↦ notation outside of records . . . 72
10.3.3 Introduction to the hidden gems ↦ (:>) and ▶(@@) . . . 72
10.3.4 The not-quite λ-expression $[x \in S \mapsto e]$ . . . 73
10.3.5 **IF THEN ELSE** . . . 73
10.3.6 **CASE ... OTHER** . . . 74

10.3.7 The winner: tuples and sequences . . . 74

11 LAB CLASS: Three_Islands 75

12 Scratch that recursion 77
12.1 Make a scratchpad for experimentation . . . 78
12.2 Playing with ▶ . . . 78
12.3 Reading the precedence / associativity table . . . 79
12.4 **ASSUME** as **assert**: "unit testing" . . . 80
12.5 A recursive function: `fact` . . . 80
12.6 Cardinality: functions vs. operators . . . 81
12.6.1 The tragedy of Darth Set-Of-All-Sets . . . 82
12.6.2 TLC vs. TLA$^+$ standard modules, **LET IN** . . . 83
12.6.3 **RECURSIVE** operators . . . 85
12.7 A straightforward `sum` . . . 85
12.8 **LAMBDA** and Fold / Reduce . . . 86

13 LAB CLASS: Scratchpad 87

14 LAB CLASS: The Worm, the Centipede, and the Grasshopper 88

15 LAB CLASS: Hard Mode: One Spec To Cross Them ALL! 89

16 LAB CLASS: Toggle Problems 91

17 TLA$^+$ CheatSheet 93

III OLD Lab classes 96

18 Preliminaries (preferably before the first lab class) 96
18.1 Setting up a work environment . . . 96
18.1.1 Operating System . . . 96
18.1.2 Choice of Linux distribution . . . 96
18.1.2.1 Provided VM: Arch-based system . . . 97
18.1.2.2 Instructions: Arch-based system . . . 97
18.1.2.3 Instructions: Debian-based system . . . 98
18.1.2.4 Instructions: Fedora / SUSE-based system . . . 98
18.1.2.5 Instructions: Microsoft's Spyware OS . . . 98
18.1.3 Check that it works, and brush up on stuff . . . 100

19 Basic finite state systems 100

20 Modelling complex systems using products 113

21 Extra exercises involving rivers (from JMC's collection) 118

22 CTL Verification 118

IV OLD Lecture Notes: Formal Verification 120

23 Meta-information about the course 123
23.1 Note on the notes: . . . 123
23.2 Course prerequisites and student assessment . . . 123

24 Introduction: What is Formal Verification? 124
24.1 Problem: Disaster stories . . . 124
24.2 Solution: Verification . . . 124
24.3 A Brief History of Program Proof . . . 125
24.4 Our focus in this course: Model Checking . . . 126
24.5 Provisional course plan . . . 126

25 State Systems and Modelisation 127
25.1 Brief Reminders About Nondeterministic Finite State Automata . . . 127
25.2 On machine: `lecture_automata_products.py` . . . 128
25.3 Modelling through automata: generalities . . . 128
25.4 Examples of Isolated Systems . . . 128
25.4.1 Digital Clock . . . 129
25.4.2 Digicode, pass 123 . . . 129
25.4.3 LIFO (Stack) and FIFO (Queue) of size 2 . . . 131
25.4.4 Incrementable Integer Variable . . . 132

26 Incrementable Unsigned Integer Variable with Overflow 133

27 Set Variable 134
27.0.1 FIFO / LIFO($\mathbf{n}, \mathbf{m}$) . . . 135
27.0.2 The Wolf, the Goat, and the Cabbage (WGC) . . . 135
27.0.3 WGC, states as functions rather than "left-bank" sets . . . 138
27.0.4 Indiana Jones and the Temple of Verification . . . 138
27.1 A Taxonomy of Automata Products . . . 140
27.1.1 Fully Synchronised Product $\otimes$ . . . 141
27.1.1.1 Fully Synchronised Product $\otimes$ for $\cap$ . . . 141
27.1.1.2 Fully Synchronised Product $\oplus$ for $\cup$ . . . 142
27.1.2 Fully Unsynchronised Product $\|$: the Shuffle . . . 142
27.1.3 Vector-Synchronised Product . . . 143
27.1.3.1 A Fully General Product . . . 145
27.1.3.2 Easy to Understand, Cumbersome to Use . . . 145
27.1.4 Named Synchronised Product . . . 146

27.1.5 Automaton Restriction . . . 147
27.2 Example Systems, Now With Some Products . . . 148
27.2.1 WGC, Now With Map-Synchronised Product . . . 148
27.2.2 Indiana Jones, now With Map-Synchronised Product . . . 151
27.2.2.1 The Solution, Concisely . . . 151
27.2.2.2 How Was I Supposed to Guess How to Handle Time? . . . 151
27.2.3 Exercise with Solution: The Bridge on the River Kwaï (FR) . . . 152
27.2.3.1 Problem Statement . . . 152
27.2.3.2 Solution . . . 153
27.2.4 Exercise with Solution: The Toggle Problems . . . 155
27.2.4.1 Problem Statement . . . 155
27.2.4.2 Solution . . . 156
27.2.5 Exercise with Solution: The Three Islands, the Two Wolves, the Goat, and the Cabbage . . . 157
27.2.5.1 Problem Statement . . . 157
27.2.5.2 Solution . . . 158
27.2.6 Exercise with Solution: Max-Weight River-Crossing Problem . . . 159
27.2.6.1 Problem Statement . . . 159
27.2.6.2 Solution . . . 160
27.2.7 Semaphores: first contact . . . 161
27.2.8 Mutable Boolean variable . . . 163
27.2.9 Sequential Programs . . . 164
27.2.9.1 Infinite Loop: `while True ...` . . . 165
27.2.9.2 `if ... else ...` . . . 166
27.2.9.3 Sequence of instructions . . . 166
27.2.9.4 Null operation: `pass` . . . 166
27.2.9.5 Application to our example . . . 166
27.2.10 Peterson's Algorithm . . . 167

28 Classical and Temporal Logics: (W)S1S, CTL*, CTL, LTL 170
28.1 Traditional Logic on Words: (w)S1S . . . 171
28.1.1 What Does that Word Salad Even Mean? . . . 172
28.1.2 Formal Syntax and Semantics . . . 174
28.1.2.1 An Example . . . 175
28.1.3 Extending the Syntax; Writing Properties . . . 175
28.1.4 The Büchi and Thatcher–Wright Theorems . . . 178
28.1.4.1 For every NFA, there is an equivalent wS1S formula . . . 178
28.1.4.2 For every wS1S formula, there is an equivalent NFA . . . 180
28.1.5 Algorithmic Complexity and Suitability for Verification . . . 180
28.1.5.1 Reminders about Complexity Theory . . . 181
28.1.5.2 What Does it Mean for Our Purposes? . . . 182
28.2 CTL*: Computation Tree and Linear Time Logic (CTL+LTL+···) . . . 183

28.2.1 Kripke Structures and Paths . . . 184
28.2.2 Syntax and Semantics of CTL* . . . 184
28.2.3 A Few Examples, to Help the Semantics go Down . . . 186
28.2.4 LTL: Linear Time Logic . . . 188
28.2.5 CTL: Computation Tree Logic . . . 189
28.2.6 Some Useful Properties and Miscellaneous Examples . . . 190
28.2.6.1 Mutual Exclusion . . . 190
28.2.6.2 Possible Access, Liveness, “infinitely often” . . . 190
28.2.6.3 Requests Get Answers, Eventually . . . 191
28.2.6.4 No Shoes, No Service . . . 191
28.2.6.5 Interdiction . . . 191
28.2.6.6 Necessary Steps . . . 192
28.2.7 Comparing LTL, CTL, and CTL* . . . 192
28.2.7.1 Expressive Powers . . . 193
28.2.7.2 Algorithmic Complexity . . . 193
28.2.8 CTL Model-Checking Algorithm . . . 193
28.3 Exercises on Logics and Automata . . . 193
28.3.1 Exam 2020–2021 . . . 193
28.3.1.1 Problem Statement . . . 193
28.3.1.2 Solution . . . 194

28 References

[Baier and Katoen, 2008] Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT press.
† Cited twice, pages 186 and 190.

[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 178.

[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 183.

[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 twice, pages 11 and 126.

[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 183.

[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 twice, pages 11 and 126.

[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 183.

[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 twice, pages 11 and 126.

[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 178.

[Thompson, 1984] Thompson, K. (1984). Reflections on trusting trust. https://dl.acm.org/doi/pdf/10.1145/358198.358210.
† Cited twice, pages 9 and 124.