Algorithmic Verification
COMP3710/6470

https://comp.anu.edu.au/courses/algorithmic-verification/
https://wattlecourses.anu.edu.au/course/view.php?id=38392

Lecturer

A/Prof Peter Höfner
CSIT (108) N234
peter.hoefner@anu.edu.au

Consultation:
Monday 1pm - 2pm
(or by appointment)

Plan/Schedule (1)

Where and When

Lectures

- Monday, 9am - 11am SRES Lecture Theatre (Geography Building 48A)
- Tuesday, 9am - 10am, SRES LectureTheatre (Geography Building 48A)

Tutorials

- Tuesday 12pm - 1pm Birch 1.29
- Tuesday 3pm - 4pm (Online)

(may change due to enrolment numbers)

Workload

- 6credits = 130 hours per term (https://policies.anu.edu.au/ppl/document/ANUP_000691)
- ≈ 10-11 hours per week for COMP3710/6470

Plan/Schedule (2)

Assessment Criteria

- Assignments: 45% (5+9+11+11+9)
- Final Exam: 55%

Assignments (tentative)

- Assignment 0:
  hand out: 26/07 hand in: 05/08
- Assignment 1:
  hand out: 09/08 hand in: 23/08
- Assignment 2:
  hand out: 23/08 hand in: 19/09
- Assignment 3:
  hand out: 20/09 hand in: 11/10
- Assignment 4:
  hand out: 11/10 hand in: 01/11

Miscellaneous

Recording
- recordings via Echo360 and Zoom.
- interactive style, plenty of examples presented
- no personal recording allowed

References (Recommended Textbooks)
- course follows more than one textbook
- all books are available in the library

- C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.
- E. Clarke, O. Grumberg and D. Peled. *Model Checking*. MIT Press, 2000.
- M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, 2004.
- B. Berard et al. *Systems and Software Verification*: Model-Checking Techniques and Tools. Springer, 2001

- other relevant literature will be given in the course

Algorithmic Verification COMP3710/6470

Lecture 1

Introduction

(last update: July 24, 2022)

Content for this lecture

1. Famous bugs in software/hardware systems
2. What is verification about?
3. What is algorithmic verification?
4. Some verification techniques and tools (overview)
5. Useful mathematical techniques/tools
6. Finite Automata

Famous Bugs

Famous Hardware Bug

- Intel P5, 1994
  - Pentium floating point unit flawed
  - Cause: missing entries in a lookup table
  - 3 to 5 million defective FPU
- Consequences
  - Intel image badly damaged
  - Replacement of defective FPUs: $US450 million

Famous Software Bugs (1)

- Therac-25, 1985–1987
  - X-ray radiation therapy machine
  - Two modes of operation: high/low energy
  - High energy requires metal shield
  - Software error: high energy generated without metal shield
- Consequences:
  - at least 5 patients died
  - Others we irradiated (overdose of radiation x100)

Famous Software Bugs (2)

- Toyota Prius, 2005
  - Sudden stalling/stopping at highway speeds
  - Cause: software error causing the program to switch to fail-safe mode
- Consequence
  - Recall of 75000 cars for software updates
  - Cost: unknown … but high

Famous Software Bugs (3)

- Ariane 5 rocket, Flight 501, 1996
  - Re-use of Ariane 4 software
  - Cause: overflow when converting 64-bits to 16-bits unsigned (horizontal part of calculation, vertical part was correct)
- Consequences
  - After 37 seconds, rocket veered off, and self-destructed
  - Cost: $US370 million

Famous Software Bugs (4)

- Northeast blackout, 2003
  - Alarm went unnoticed
  - Bug in the alarm system
  - Cause: probably race condition
- Consequences:
  - Power failure from 4pm to 11pm (up to 2 days)
  - 55 million people affected
  - Cost: more than $US6 billion

What is Verification?

Software Verification

Software verification is a discipline of software engineering whose goal is to assure that software fully satisfies all the expected requirements. (wikipedia)

- Requirements R
  - What the system is expected to do
- Software S (note: same applies for hardware)
  - A set of interacting processes (programs)
- Satisfaction
  - Check that S satisfies R

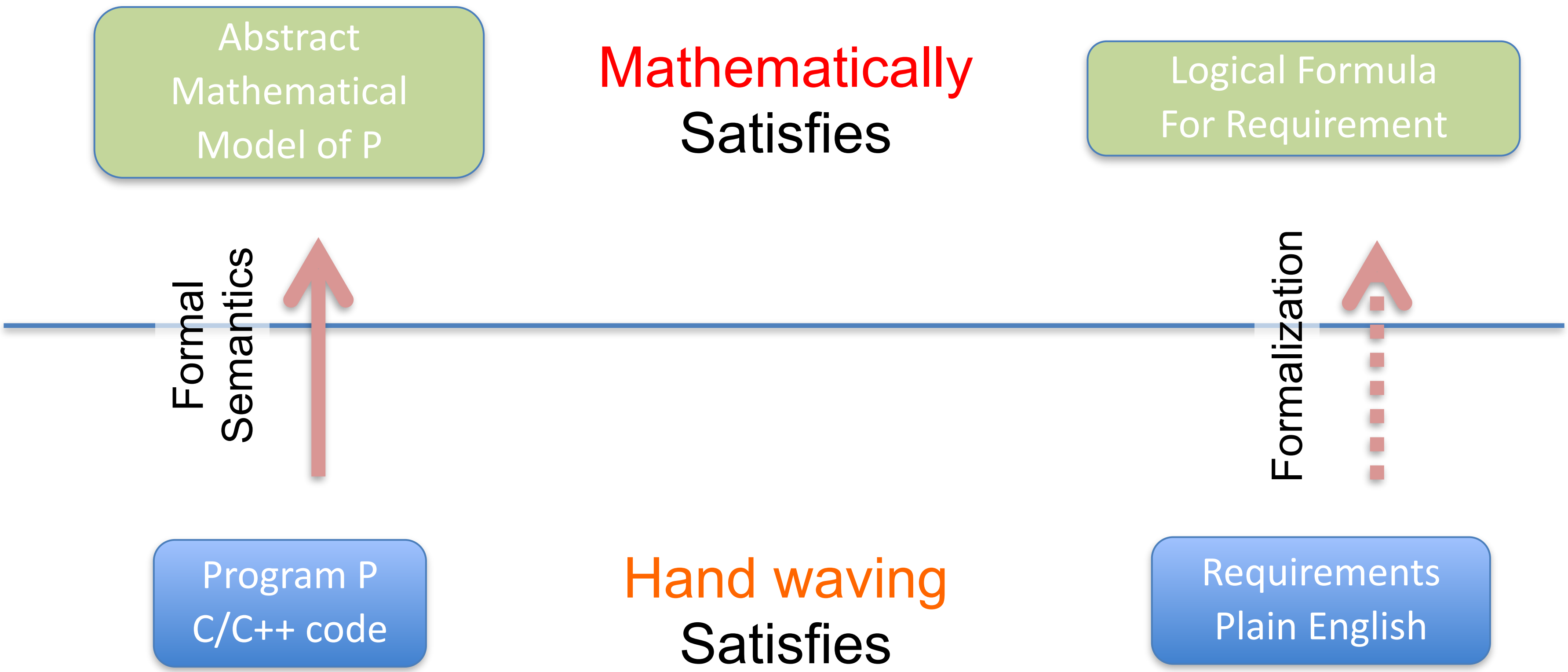
The Need for Verification

- Computers are increasingly complex
  - Difficult to prove “correctness” of hardware
  - Difficult to prove correctness of software
    - Programs with millions of lines of codes
    - Concurrent programs (threads)
- Software/hardware controlled systems are everywhere, some safety-critical
  - Complex interaction of processes (concurrent systems)
  - Certification mandatory (e.g. aircrafts software)
  - Subject to attacks (e.g. banks, e-voting…)

Approaches to "Verification"

- Testing:
  - Build the software
  - Run experiments (infinitely many …)
  - Drawback: not exhaustive
  - Advantages: only requires the software (and tests)
- Are there better approaches?
  - Rigorous?
  - Exhaustive?
  - Automatic?

The Big Picture



Requirements for Formal Verification

- Software Verification e.g. C/C++ programs
  - Goal: Rigorous proof that the system satisfies the requirements
- Requires:
  - Clear/unambiguous requirements R
  - A Program P
  - Clear/unambiguous semantics of P:
  - Proof techniques to derive proofs/theorems like

$$[\![P]\!] \models R$$

What is Algorithmic Verification?

Formal Verification Techniques

Technique	Complexity of requirements	Scalability	Automation	Speed	Confidence
Mathematical proof (by hand)	Unbounded	Low	None	Low	Low/*High*
Theorem Provers	Unbounded	Medium	Medium	Medium	High
Model Checking	Medium	High	Full	Very High	High
Static Analysis	Medium	High	Full	Very High	High

Algorithmic / Automated Verification

Brief (and biased) History of Model Checking and Static Analysis

Model Checking (1980)

- Introduced independently by Clarke, Emerson, Sistla, and Queille and Sifakis in 1980

Turing Award 2007

- Formal Model: finite state machine S
- Requirements: Temporal logics R
  - Introduced by Pnueli, 1977

Turing Award 1996

- Model Checking algorithm:
- Check dynamic requirements on executions
  - Automatically verifies that S satisfies R

    If not gives a counter example
  - Exhaustive search of state space

Large system: State Explosion Problem

State Explosion Problem

- Program with 100 integer variables within [0,10]
  - 10^{100} states
  - “size” of the universe: 10^{78} atoms
- 100 concurrent components, 10 states each
  - 10^{100} states
- How to combat state explosion problem:
  - Symbolic model checking, McMillan, 1987
  - Abstraction/Refinement, Clarke, 1990
  - SAT-based techniques, 2000
  - Compositional Approaches such as Rely/Guarantee or Owicki-Gries

State Explosion Problem

- Program with 100 integer variables within [0,10]
  - 10^{100} states
  - "size" of the universe: 10^{78} atoms
- 100 concurrent components, 10 states each
  - 10^{100} states
- How to combat state explosion problem:
  - Symbolic model checking, McMillan, 1987
  - Abstraction/Refinement, Clarke, 1990
  - SAT-based techniques, 2000
  - Compositional Approaches such as Rely/Guarantee or Owicki-Gries

Model Checking Tools (1)

- SMV, McMillan, 1987
  NuSMV, Cimatti, 2002
  nuXmv, 2014
- Symbolic: processes symbolic representation of sets of states
- Data structure: Binary Decision Diagrams
- Used in hardware verification

Model Checking Tools (2)

- SPIN, Holtzmann, 1990
- Explicit model checker
- Check C-like communicating programs
  - Promela language
- Heuristics to control state explosion
  - Partial order reduction techniques
  - Hashing
- Used in protocol verification

Model Checking Tools (3)

- SAT-based tools
- Bounded model-checking
  - Executions of length bounded by integer N
  - Encode model checking as a SAT problem
- SLAM, T. Ball and S. Rajamani, Microsoft, 2000
  - Verifies drivers
- C-BMC, Kroenig, 2002
  - Verifies ANSI-C programs

Static Analysis

- Check static requirements on program
  - E.g. no NULL pointer dereference
  - No Array out-of-bound access
- Compute (loop) invariants
- Based on Abstract Interpretation, Cousot & Cousot, 1977
  - Control flow analysis
  - Data flow analysis

Static Analysis Tools

- ASTREE, Absint, 2003
- Coverity, 2003
- Grammatech, 2005
- Polyspace
- PVS-Studio
- Goanna, 2009
- Skink, 2015

Algorithmic Verification COMP3710/6470

Lecture 1

Modelling Systems

Useful Mathematical Tools

Set Theory

- Union: $A \cup B$
- Intersection: $A \cap B$
- Empty set: $\varnothing$
- Set difference: $A \setminus B$
- Complement: $\overline{A}$
- Distributivity:

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

Functions

$$f : A \to B$$

$$X \subseteq A, f(X) = \{f(x), x \in X\}$$

$$Y \subseteq B, f^{-1}(Y) = \{x \in A, f(x) \in Y\}$$

Relations

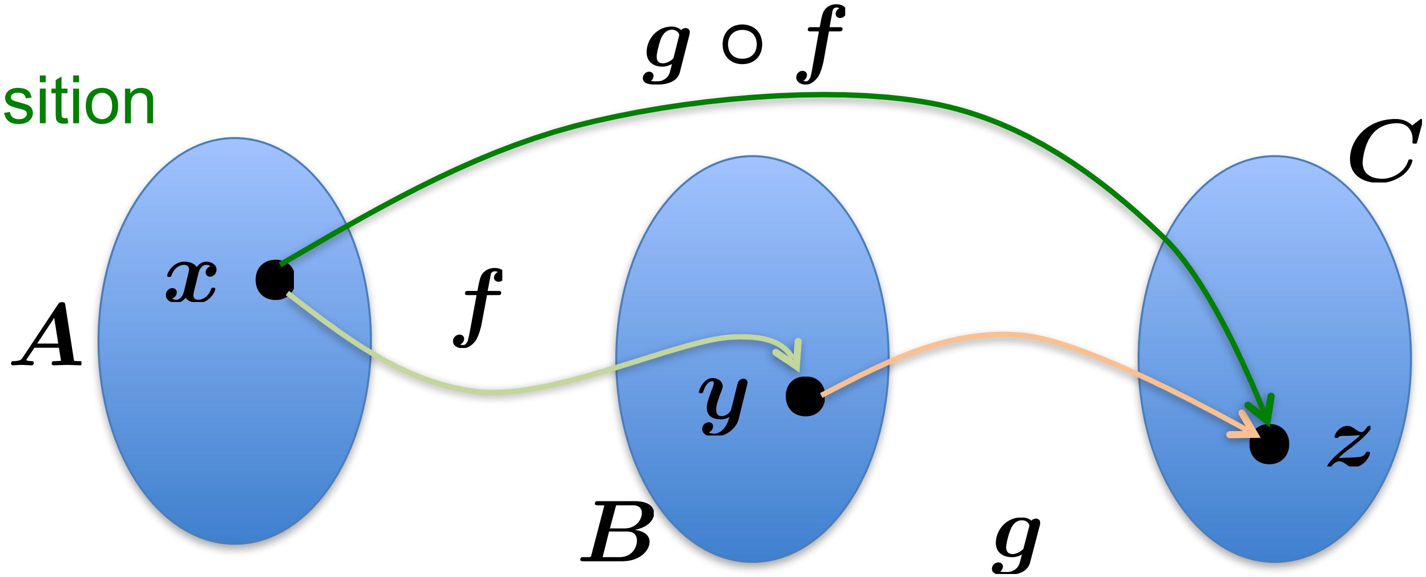
$$\mathcal{R} \subseteq E \times E$$

$(e, e') \in \mathcal{R}$ *or* $e \, \mathcal{R} \, e'$

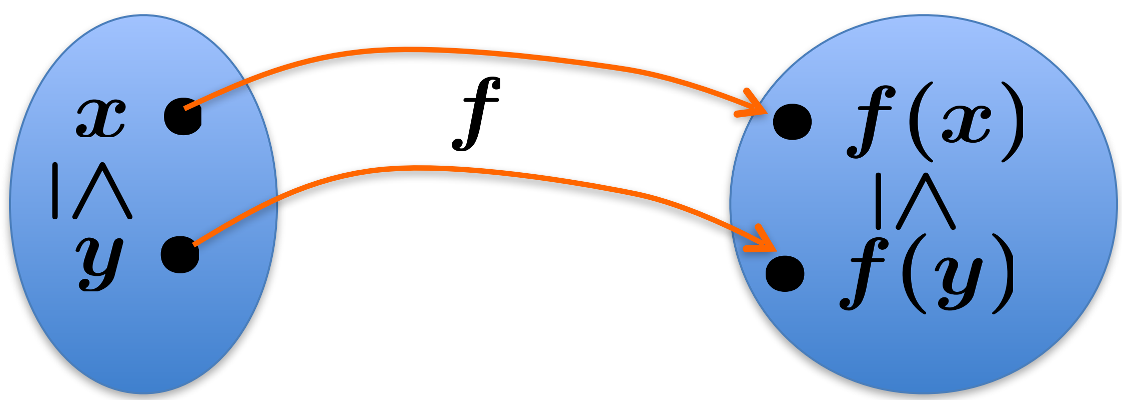
$$e' \, \mathcal{R}^{-1} \, e \iff e \, \mathcal{R} \, e'$$

More on Functions

Composition



Monotonicity



Propositional Logics

- set of atomic propositions $\mathcal{P} = \{a, b, c \cdots, \}$
- Formulas
  - each $p \in \mathcal{P}$ is a formula,
  - if P, Q are formulas, $P \wedge Q$, $P \vee Q$ and $\neg Q$ are formulas
- Valuation $\mathcal{V} : \mathcal{P} \longrightarrow \{True, False\}$
- Truth-value $[P](\mathcal{V})$ of a formula P for valuation $\mathcal{V}$:
  - $p \in \mathcal{P}$, $[p](\mathcal{V}) = \mathcal{V}(p)$,
  - if P, Q are formulas, $\bowtie \in \{\wedge, \vee\}$,
    $$[P \bowtie Q](\mathcal{V}) = [P](\mathcal{V}) \bowtie [Q](\mathcal{V}),$$
  - $[\neg Q](\mathcal{V}) = \neg[Q](\mathcal{V})$

Semantics

Propositional Logic

Set of atomic propositions $\mathcal{P} = \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \cdots, \}$

Syntax of propositional formulas over $\mathcal{P}$

- ***True*** is a formula,
- each $\boldsymbol{x} \in \mathcal{P}$ is a formula,
- if $\boldsymbol{P}, \boldsymbol{Q}$ are formulas, $\boldsymbol{P} \wedge \boldsymbol{Q}$ and $\neg \boldsymbol{Q}$ are formulas

Semantics: given $\mathcal{E} \subseteq \mathcal{P}$ an environment, define $\models$

- $\mathcal{E} \models \textit{\textbf{True}}$,
- $\boldsymbol{x} \in \mathcal{P}, \mathcal{E} \models \boldsymbol{x} \iff \boldsymbol{x} \in \mathcal{E}$
- given $\boldsymbol{P}, \boldsymbol{Q}$, $\mathcal{E} \models \boldsymbol{P} \wedge \boldsymbol{Q} \iff \mathcal{E} \models \boldsymbol{P}$ and $\mathcal{E} \models \boldsymbol{Q}$
- given $\boldsymbol{P}$, $\mathcal{E} \models \neg \boldsymbol{P} \iff$ not $\mathcal{E} \models \boldsymbol{P}$ ($\mathcal{E} \not\models \boldsymbol{P}$).

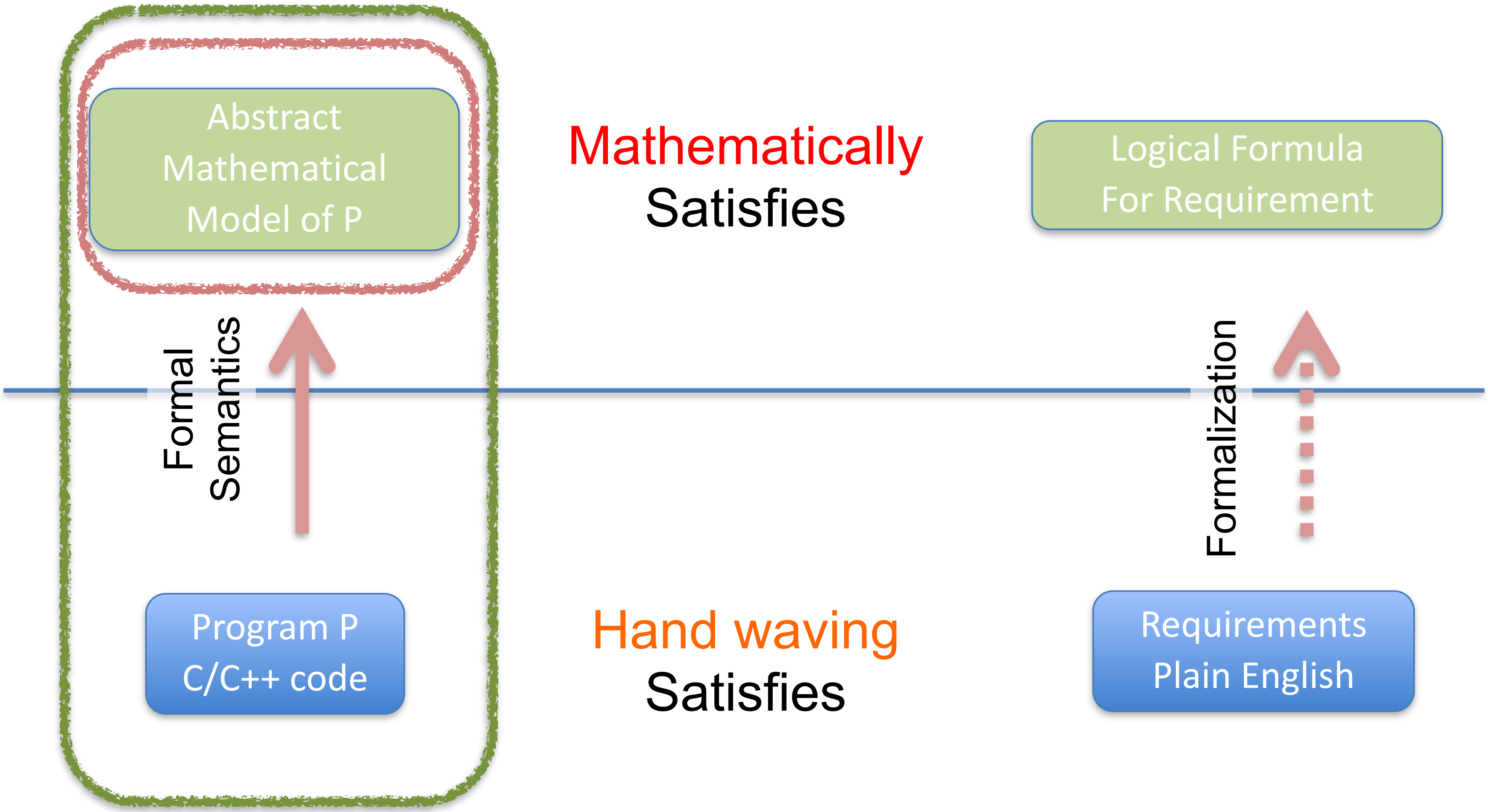
Examples

Atomic propositions $\mathcal{P} = \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$
Context $\mathcal{E} = \{\boldsymbol{a}, \boldsymbol{b}\}$

- $\mathcal{E} \models \neg(\boldsymbol{a} \wedge \boldsymbol{b})$?
- $\mathcal{E} \models \boldsymbol{b}$?
- $\mathcal{E} \models \neg\boldsymbol{b}$?
- $\mathcal{E} \models (\boldsymbol{a} \wedge \boldsymbol{b}) \wedge (\neg(\boldsymbol{b} \wedge (\neg\boldsymbol{c})))$?

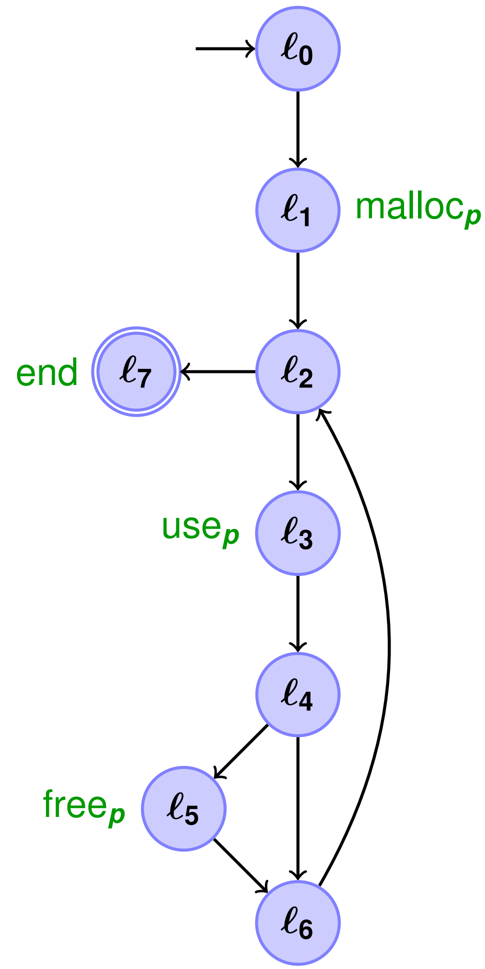
Define $\boldsymbol{a} \vee \boldsymbol{b}$, $\boldsymbol{a} \implies \boldsymbol{b}$ and $\boldsymbol{a} \iff \boldsymbol{b}$

The Big Picture



C Program Example

```
   void foo() {
0    int x, a;
1    int *p = malloc(sizeof(int));
2    for (x = 10; x > 0; --x) {
3      a = *p;
4      if (x <= 1)
5        free(p);
6    }
```



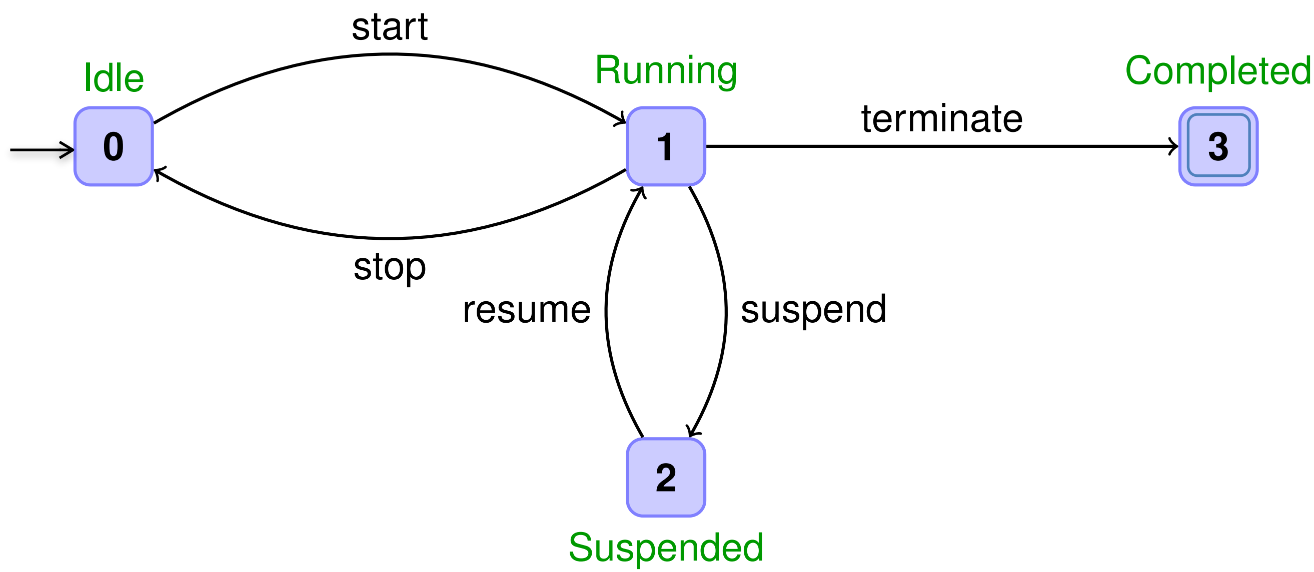
$$\ell_0 \longrightarrow \ell_1 \longrightarrow \ell_2 \longrightarrow \ell_3 \longrightarrow \ell_4 \longrightarrow \ell_6 \longrightarrow \ell_2$$
$$\longrightarrow \ell_3 \longrightarrow \ell_4 \longrightarrow \ell_6 \longrightarrow \ell_2 \longrightarrow \ell_7$$

$$\emptyset \longrightarrow \{\mathit{malloc}_p\} \longrightarrow \emptyset \longrightarrow^* \{\mathit{use}_p\}$$
$$\longrightarrow^* \{\mathit{use}_p\} \longrightarrow^* \{\mathit{end}\}$$

Task/Thread Example

Task, process or thread

- Can be in state idle, running, suspended, completed
- Can be scheduled, preempted, added to the queue



$$\mathbf{0} \xrightarrow{\textbf{\textit{start}}} \mathbf{1} \xrightarrow{\textbf{\textit{suspend}}} \mathbf{2} \xrightarrow{\textbf{\textit{resume}}} \mathbf{1} \xrightarrow{\textbf{\textit{terminate}}} \mathbf{3}$$

0 1 2 1 2 1 2 1 3

start* (*suspend resume*)* *terminate

Labeled Automata

Given: a set of atomic propositions $\mathcal{P} = \{a, b, c \cdots ,\}$

A labeled automaton (LA) A is a tuple $(Q, q_0, \Sigma, \delta, F, L)$ where:

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- Σ is a finite alphabet of actions,
- $\delta : Q \times \Sigma \longrightarrow 2^Q$ is the transition relation,
- $F \subseteq Q$ is a set of final states,
- $L : Q \longrightarrow 2^{\mathcal{P}}$, a labeling function.

if $|\Sigma| = 1$, actions are omitted and $\delta : Q \longrightarrow 2^Q$

Kripke structure (Q, q_0, δ, L)
$|\Sigma| = 1$, $F = Q$ and $\forall s \in Q, \delta(s) \neq \varnothing$.

Runs

A run ϱ of A is a sequence

$$\varrho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_n$$

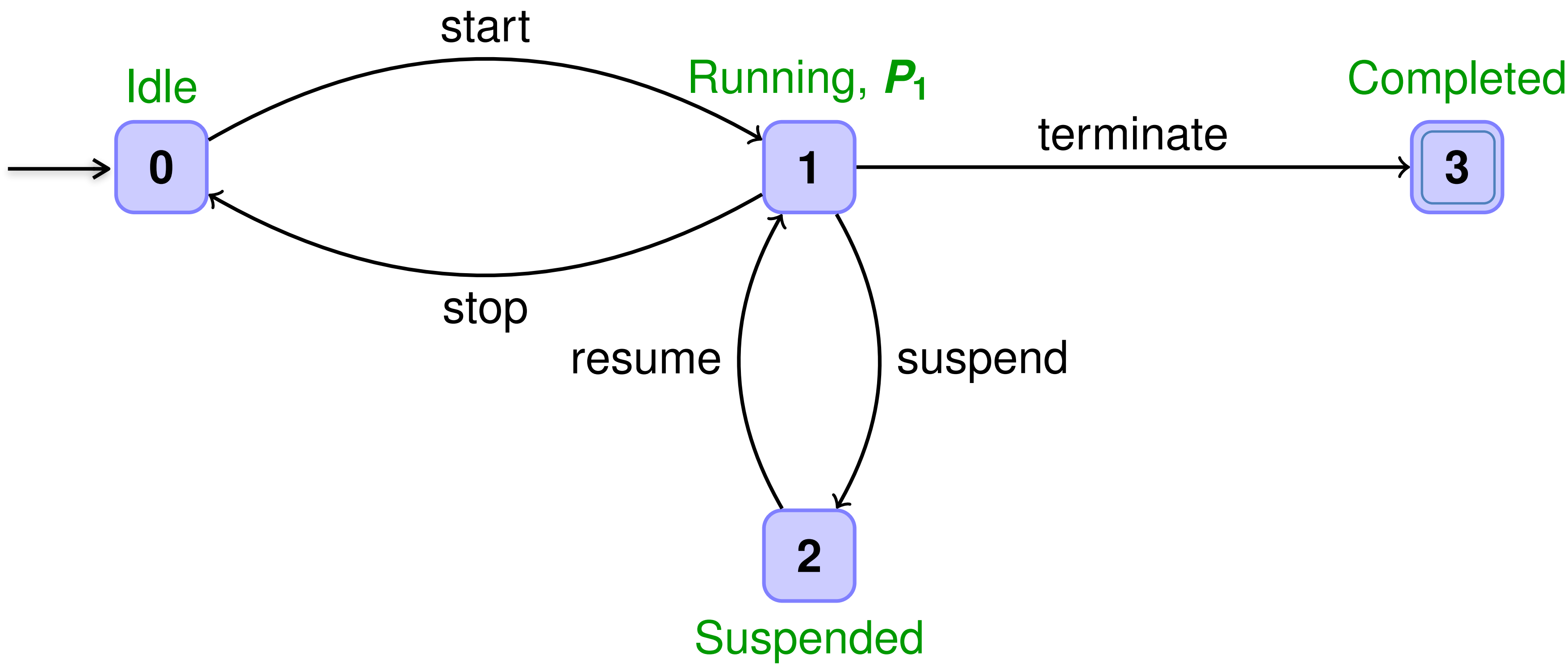
with $s_{i+1} \in \delta(s_i, a_i), 0 \leq i \leq n-1$.

$Runs(s_0)$: set of runs of A from s_0

with each run we can associate
a sequence of sets of atomic propositions

$$L(s_0) \longrightarrow L(s_1) \longrightarrow L(s_2) \longrightarrow \cdots \longrightarrow L(s_n)$$

Scheduling Example



$\mathbf{0} \longrightarrow \mathbf{1} \longrightarrow \mathbf{2} \longrightarrow \mathbf{1} \longrightarrow \mathbf{2}$

$\{\textit{Idle}\} \longrightarrow \{\textit{Running}, P_1\} \longrightarrow \{\textit{Suspended}\} \longrightarrow \{\textit{Running}\} \longrightarrow \{\textit{Running}\}$

Semantics of Automata

Set of Runs and Computation Tree

Semantics = Set of Runs

- Events/States have one successor in each run
- Linear (Time) semantics of ***A***:

$$[\![A]\!] = Runs(q_0)$$

Semantics is an infinite set of (finite) runs

Semantics = Computation Tree

Assume $\boldsymbol{A}$ is a Kripke Structure.

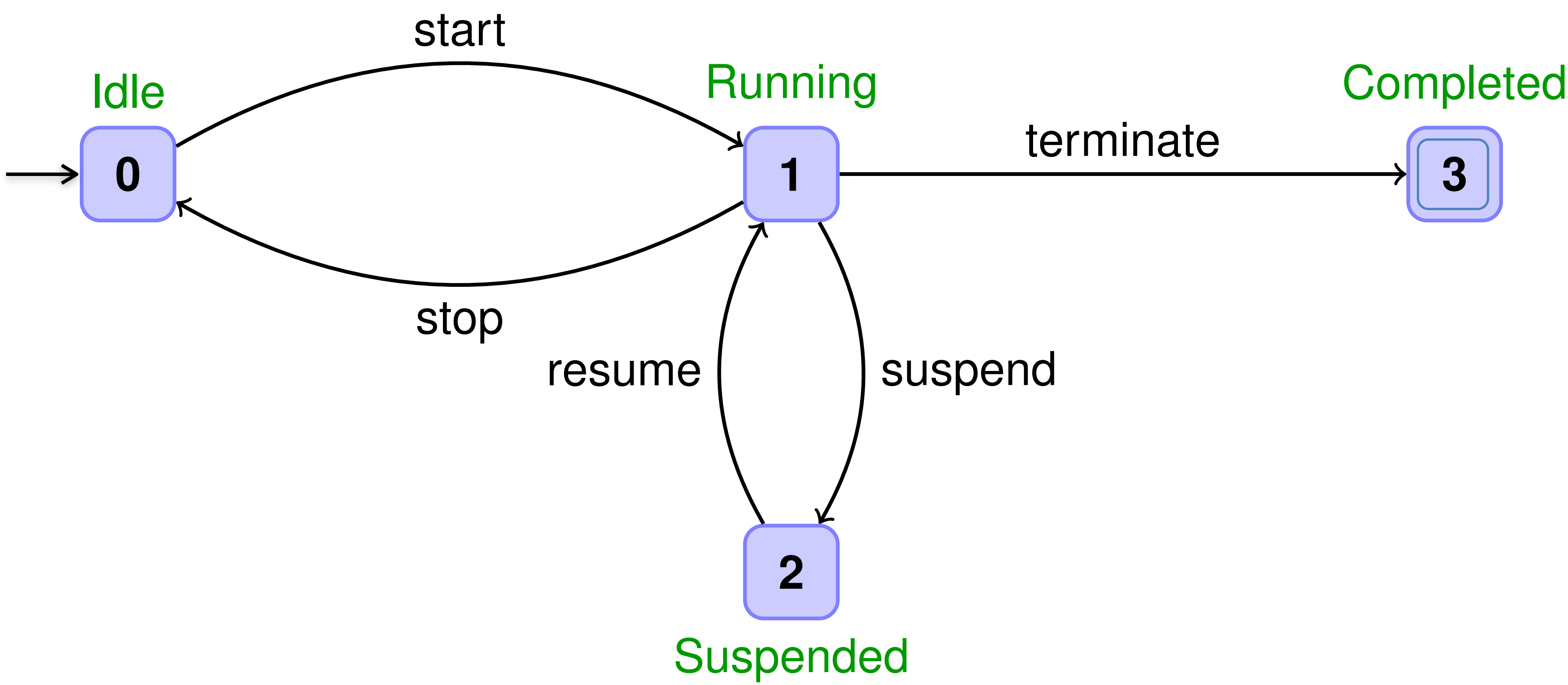
- States have many successors in the computation tree
- Branching (Time) semantics of $\boldsymbol{A}$.

$[\![\boldsymbol{A}]\!]$ is the labeled tree defined by:

- the root $\boldsymbol{r}$, labelled by $\boldsymbol{q_0}$;
- if $\boldsymbol{n}$ is a vertex of the tree, labelled by $\boldsymbol{s}$, then
  1. if $\boldsymbol{s}' \in \delta(\boldsymbol{s})$, then $\boldsymbol{n} \longrightarrow \boldsymbol{n}\ \boldsymbol{s}'$ is an edge in the tree equivalently $\boldsymbol{n}\ \boldsymbol{s}'$ is a child of $\boldsymbol{n}$,
  2. $\boldsymbol{n}\ \boldsymbol{s}'$ is labeled by $\boldsymbol{s}'$.

Exercise

Build the (beginning) of the tree for the scheduling example.



Automata Theory (1)

- Alphabet Σ: finite set of letters.
- Word: finite sequence of letters.
- ε: the empty word.
- Σ^*: set of words on Σ (including ε)

A Finite Automaton (FA) A is a tuple $(Q, q_0, \Sigma, \delta, F)$ where:

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- Σ is a finite alphabet of actions,
- $\delta : Q \times \Sigma \longrightarrow 2^Q$ is the transition relation,
- $F \subseteq Q$ is a set of final states.

A is deterministic if $\forall(s, a) \in Q \times \Sigma, |\delta(s, a)| \leq 1$.
Otherwise A is non-deterministic.

Automata Theory (2)

A run of A from s_0 is a sequence of transitions

$$s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} \cdots \xrightarrow{l_{n-1}} s_{n-1} \xrightarrow{l_n} s_n$$

Notation: $s_0 \xrightarrow{l_1\ l_2\ \cdots l_n} s_n$ and $s \xrightarrow{*} s'$

The language, $\mathcal{L}(A) \subseteq \Sigma^*$, accepted/recognized by A is:

$$\mathcal{L}(A) = \{w \in \Sigma^*, q_0 \xrightarrow{w} q, q \in F\}$$

Automata Theory (3)

A finite automaton with ε-transitions is a FA with $\varepsilon \in \Sigma$.

Classes of Finite Automata:

- DFA: deterministic finite automata
- NFA: non-deterministic finite automata
- NFA$^\varepsilon$: finite automata with ε transitions.

Important result: w.r.t. language generation/acceptance

$$\text{DFA} \equiv \text{NFA} \equiv \text{NFA}^\varepsilon$$

Determinization: subset construction

References

- Propositional Logic:
  - Huth/Ryan: Logic in Computer Science Section 1
  - Bayer/Katoen: Principles of Model Checking Appendix A3
- Automata:
  - Sipser: Intro. to the Theory of Computation Parts of 1.1 and 1.2
  - Kozen: Automata and Computability Sections 3-5