Introduction to Computer Systems Verification

Bernard Boigelot

E-mail : bernard.boigelot@uliege.be
WWW : https://people.montefiore.uliege.be/boigelot/
https://people.montefiore.uliege.be/boigelot/courses/verif

Main reference: Pierre Wolper, Francqui Chair lectures, 1998.

Chapter 1

Verification: Goals and principles

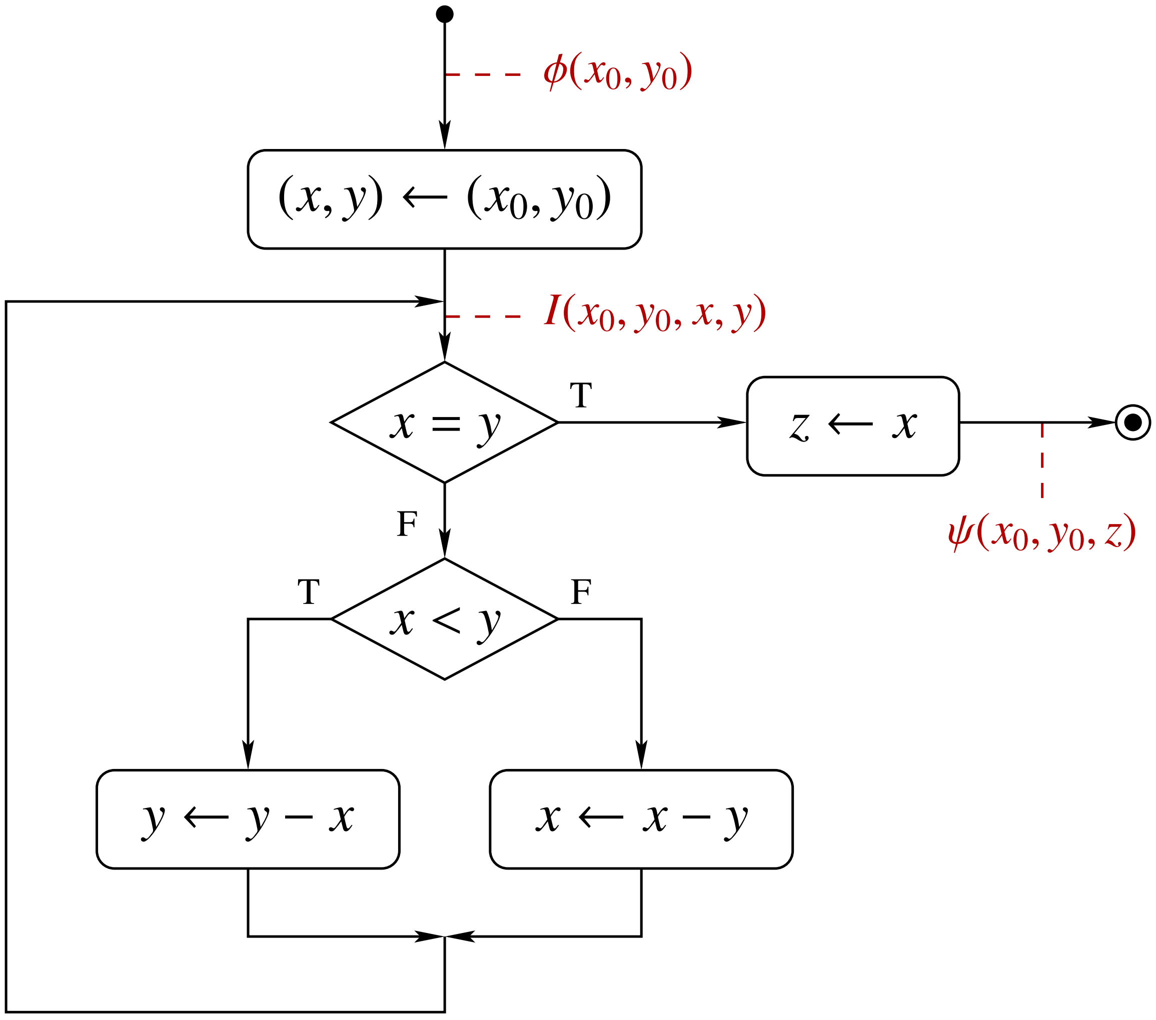
Introduction

Problem: What does this function compute (assuming that initially, x, y > 0)?

```
int f(int x, int y)
{
  while (x != y)
    if (x < y)
      y -= x;
    else
      x -= y;

  return x;
}
```

Answer: `f(`x_0`, `y_0`)` computes the *greatest common divisor (gcd)* of x_0 and y_0, and we can prove it!



- Initially:

$$\phi(x_0, y_0) : x_0 > 0 \ \wedge \ y_0 > 0$$

- At each iteration:

$$\begin{aligned} I(x_0, y_0, x, y) : x_0 > 0 \ \wedge \ y_0 > 0 \ \wedge \ x > 0 \ \wedge \ y > 0 \\ \wedge \ \gcd(x, y) = \gcd(x_0, y_0) \end{aligned}$$

  (Indeed, if $a > b > 0$, then $\gcd(a, b) = \gcd(b, a) = \gcd(a - b, b)$.)

- At the end:

$$\psi(x_0, y_0, z) : z = \gcd(x_0, y_0)$$

Notes:

- These properties (which express an invariant of the program) can be proved using Hoare logic.

- One can also prove (using a different technique) that this program terminates for all $x_0, y_0 > 0$.

Question: Can we really analyze every program using this method?

Answer: No, for the following reasons.

- From a theoretical point of view, the problem is undecidable.
- Finding invariants is difficult, especially for programs that have already been written.

  Note: Once invariants have been found, tools exist to help to prove them.
- Hoare logic is not easily applicable to some mechanisms of real programming languages, such as pointers, exceptions, dynamic memory allocation, . . .
- Concurrent (parallel) programs are especially tricky to handle: Hoare logic reasons about the initial and final states of computations, and parallelism requires to take into account what happens during them.
- Checking programs manually is tedious and error-prone.

Software verification

Goal: Check algorithmically whether a given program is correct, i.e., whether all its executions satisfy some properties expressed by a specification.

Examples of properties:

- The function `f` always computes the gcd of its arguments.
- For every access to an array, the index stays within the bounds of this array.
- Two processes can never enter simultaneously a critical section.
- Every request sent to a server will eventually be answered.
- ...

Main principles

- The program to be verified is written in an abstract formalism, that precisely describes its relevant elements, and abstracts away unnecessary details.
- The specification is expressed as a formula, in a logic that makes it possible to reason about the behavior of executions.
- The goal is to check algorithmically whether every execution of the program satisfies the specification. Since in logic, this is equivalent to checking that the program is a model of the formula expressing the specification, this approach is called model checking.

Note: This is similar to other engineering disciplines (civil engineering, electronics, aeronautics, . . .), where designs are also checked by computer.

Target application

The focus of verification is mainly on concurrent reactive systems, i.e., systems composed of several processes that continuously or frequently interact with their environment.

Examples:

- Elements of concurrent programs (algorithms for mutual exclusion, synchronization, leader election, . . .).
- Programs interacting with a physical system.
- Process control software.
- Embedded controllers.
- Communication protocols.
- . . .

Justification:

- These systems are usually difficult to test: large number of possible behaviors, non reproducible executions, unpredictable environment, . . .
- Yet they are used in safety-critical applications, where failure can lead to loss of life or money.

  Famous examples:

  - Therac-25 radiation therapy machine (several deaths).
  - Ariane 5 maiden flight (> 300 M€).
  - Mars Polar Lander and Mars Climate Orbiter (> 300 M€).
  - . . .
- They are usually simple, with little data manipulation and finite-state descriptions.

Writing specifications

Problem: Obtaining an adequate, complete and correct specification is difficult (sometimes as difficult as writing the program itself).

This is not a big deal:

- Checking simple properties (e.g., the program can never reach the state *#error*), as well as generic ones (absence of buffer overflows, of deadlocks, . . .) is already extremely useful.

- The goal is not really to prove that the system is completely, absolutely and undoubtedly correct, but to have powerful tools for finding bugs and improve the confidence in the design.

Notes:

- The correct behavior of a system does not only depend on its software, but also on its CPU, OS, compiler, . . .

- Verification is a tool that accompanies other strategies for developing good-quality software: good methodology, testing, documentation, ...
- We have mentioned concurrent reactive systems, but there are other important application domains, such as the verification of hardware designs (famous example: Pentium FDIV bug, > 400 M€).

What follows in the course:

- How to write programs to be verified.
- How to write specifications, using tools such as finite automata and temporal logic.
- How to explore algorithmically the state space of a program.
- How to deal with large, or even infinite, state spaces.

Chapter 2

Modeling concurrent reactive systems

Motivation

There exist many programming languages, and their semantics is not always precisely defined.

Example (C language): What is the effect of the following code?

```
int x = 0;
x = ++x + x--;
```

To avoid this issue, we introduce a modeling language, with the following characteristics:

- It has a precise semantics.
- It is simple enough to be handled by verification algorithms.
- It is expressive enough to describe the programs of interest.
- It can model concurrency, as well as common process communication and synchronization algorithms.

Formal Concurrent Systems

The language of Formal Concurrent Systems (FCS) is a modeling formalism with the following features:

- The number of processes is fixed and constant.
- Each process has finitely many control locations.
- The processes access a commonly shared memory, represented by a fixed number of variables.
- The actions of the system are described by transitions, that specify
  - A change of control location for one (simple transition) or several (joint transition) processes.
  - A condition (guard) on the variable values. This condition must be true for the transition to be enabled.
  - A set of simultaneous memory assignments that describe a modification of the memory content.

Definition

A Formal Concurrent System is defined by a triple $(\mathcal{P}, \mathcal{M}, \mathcal{T})$, where

- $\mathcal{P}$ is a finite set of processes.

  Each process $p_i \in \mathcal{P}$ is characterized by a finite set $\ell(p_i)$ of control locations, with an initial location $\ell_0^{p_i} \in \ell(p_i)$. The sets of locations of distinct processes must be disjoint: for all $i \neq j: \ \ell(p_i) \cap \ell(p_j) = \emptyset$.

- $\mathcal{M}$ is a memory, consisting of a finite set of variables and a function $\mathcal{I}$ that assigns an initial value to each variable.

  Note: The type of the variables is not imposed. In this course, we will only use common simple types: integers, Booleans, arrays, ...

- $\mathcal{T}$ is a finite set of transitions.

A transition $t \in \mathcal{T}$ is characterized by the following elements:

- A function $n(t)$ that gives a name (in the form of an identifier) to t.
- The set $p(t) \subseteq \mathcal{P}$ of the processes that are active for the transition.
- For each process $p_i \in p(t)$:
  - A source location $\ell_s(t, p_i) \in \ell(p_i)$.
  - A destination location $\ell_d(t, p_i) \in \ell(p_i)$.
- A Boolean condition $C(t)$ expressed over the values of the variables in $\mathcal{M}$.

  Note: The form of conditions is not imposed. It can be any Boolean combination of computable predicates.
- An assignment $A(t)$ of the form

$$(var_1, var_2, \ldots, var_k) \ := \ (exp_1, exp_2, \ldots, exp_k),$$

  where $k \geq 0$, $var_1, \ldots, var_k \in \mathcal{M}$ such that $var_i \neq var_j$ for all $i \neq j$, and $exp_1, \ldots, exp_k$ are computable expressions over the value of the variables in $\mathcal{M}$.

Note: When the transition is followed, the assignment

$$(var_1, var_2, \ldots, var_k) \ := \ (exp_1, exp_2, \ldots, exp_k)$$

is performed in two steps:

1. First, all the expressions exp_1, exp_2, ..., exp_k are evaluated.

2. Then, the values of var_1, var_2, ..., var_k are updated.

Notation conventions

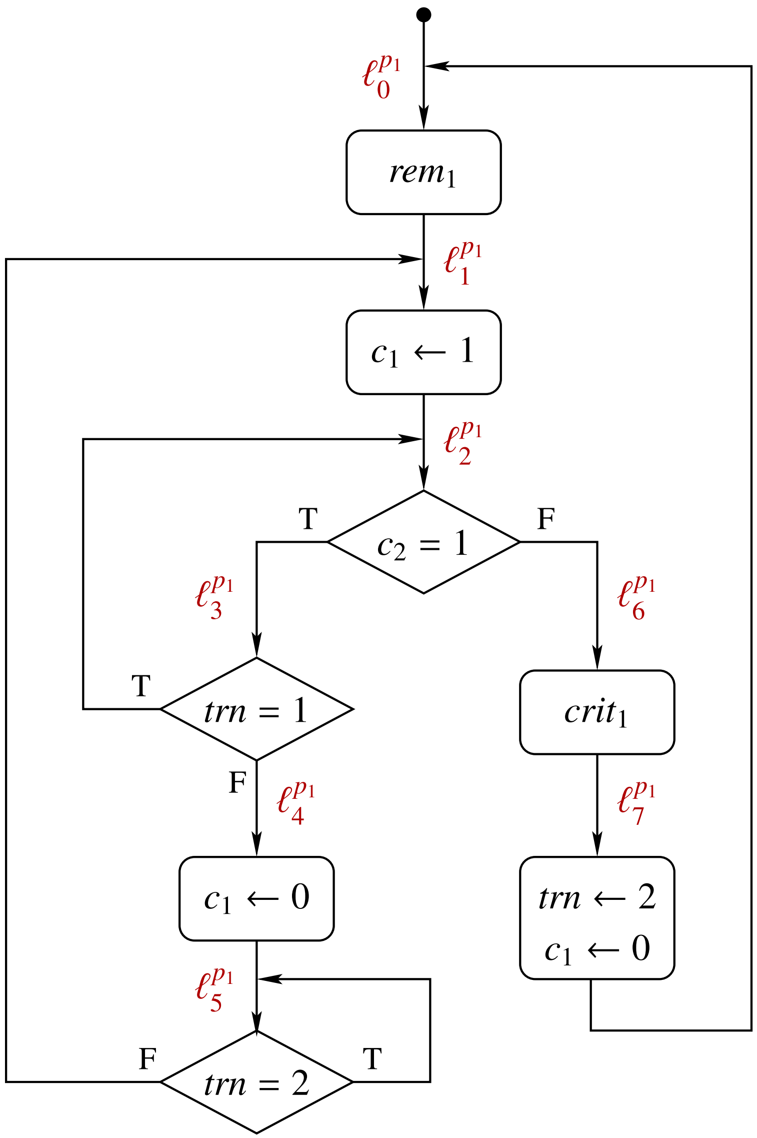
- Processes in $\mathcal{P}$ are denoted by $p_1, p_2, p_3, \ldots$
- The control locations of a process p_i are denoted by $\ell_0^{p_i}, \ell_1^{p_i}, \ell_2^{p_i}, \ldots$ By convention, $\ell_0^{p_i}$ is the initial location of p_i.
- The variables in $\mathcal{M}$ are denoted by $x_1, x_2, x_3, \ldots$, or by specific identifiers.
- A transition t is denoted by

$$n(t),\ p(t) :\ (sources,\ C(t) \rightarrow A(t),\ destinations),$$

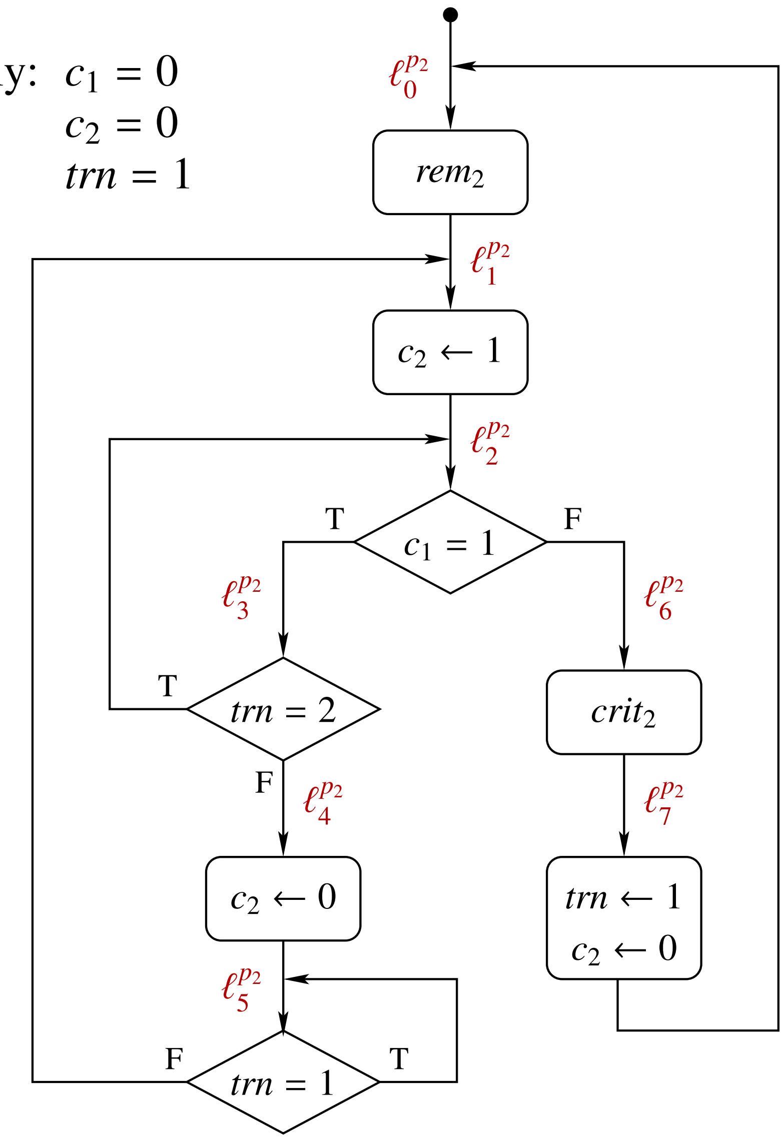
  where

  - $sources$ is a list of the source locations $\ell_s(t, p_i)$, for all $p_i \in p(t)$.
  - $destinations$ is a list of the destination locations $\ell_d(t, p_i)$, for all $p_i \in p(t)$.

Example: Dekker's mutual exclusion algorithm



Initially: $c_1 = 0$
$c_2 = 0$
$trn = 1$



Dekker's algorithm as a FCS

- Processes: $\mathcal{P} = \{p_1, p_2\}$, with

$$\begin{aligned}\ell(p_1) =& \{\ell_0^{p_1}, \ell_1^{p_1}, \ell_2^{p_1}, \ell_3^{p_1}, \ell_4^{p_1}, \ell_5^{p_1}, \ell_6^{p_1}, \ell_7^{p_1}\}\\ \ell(p_2) =& \{\ell_0^{p_2}, \ell_1^{p_2}, \ell_2^{p_2}, \ell_3^{p_2}, \ell_4^{p_2}, \ell_5^{p_2}, \ell_6^{p_2}, \ell_7^{p_2}\}\end{aligned}$$

- Memory: $\mathcal{M} = \{c_1, c_2, trn\}$, with

$$\mathcal{I}(c_1) = 0,\ \mathcal{I}(c_2) = 0, \mathcal{I}(trn) = 1$$

- Transitions: The elements of $\mathcal{T}$ are:

$$\begin{aligned}
rem_1,\ p_1 &: \left(\ell_0^{p_1},\ \text{true} \rightarrow (\,)\ :=\ (\,),\ \ell_1^{p_1}\right)\\
t_{1,2},\ p_1 &: \left(\ell_1^{p_1},\ \text{true} \rightarrow (c_1)\ :=\ (1),\ \ell_2^{p_1}\right)\\
t_{1,3},\ p_1 &: \left(\ell_2^{p_1},\ c_2 \neq 1 \rightarrow (\,)\ :=\ (\,),\ \ell_6^{p_1}\right)\\
crit_1,\ p_1 &: \left(\ell_6^{p_1},\ \text{true} \rightarrow (\,)\ :=\ (\,),\ \ell_7^{p_1}\right)\\
t_{1,5},\ p_1 &: \left(\ell_7^{p_1},\ \text{true} \rightarrow (trn,\ c_1)\ :=\ (2,\ 0),\ \ell_0^{p_1}\right)\\
t_{1,6},\ p_1 &: \left(\ell_2^{p_1},\ c_2 = 1 \rightarrow (\,)\ :=\ (\,),\ \ell_3^{p_1}\right)\\
t_{1,7},\ p_1 &: \left(\ell_3^{p_1},\ trn = 1 \rightarrow (\,)\ :=\ (\,),\ \ell_2^{p_1}\right)
\end{aligned}$$

$$
\begin{aligned}
t_{1,8},\ p_1 &: \left(\ell_3^{p_1},\ trn \neq 1 \to (\,) := (\,),\ \ell_4^{p_1}\right)\\
t_{1,9},\ p_1 &: \left(\ell_4^{p_1},\ \textsf{true} \to (c_1) := (0),\ \ell_5^{p_1}\right)\\
t_{1,10},\ p_1 &: \left(\ell_5^{p_1},\ trn = 2 \to (\,) := (\,),\ \ell_5^{p_1}\right)\\
t_{1,11},\ p_1 &: \left(\ell_5^{p_1},\ trn \neq 2 \to (\,) := (\,),\ \ell_1^{p_1}\right)\\
rem_2,\ p_2 &: \left(\ell_0^{p_2},\ \textsf{true} \to (\,) := (\,),\ \ell_1^{p_2}\right)\\
t_{2,2},\ p_2 &: \left(\ell_1^{p_2},\ \textsf{true} \to (c_2) := (1),\ \ell_2^{p_2}\right)\\
t_{2,3},\ p_2 &: \left(\ell_2^{p_2},\ c_2 \neq 1 \to (\,) := (\,),\ \ell_6^{p_2}\right)\\
crit_2,\ p_2 &: \left(\ell_6^{p_2},\ \textsf{true} \to (\,) := (\,),\ \ell_7^{p_2}\right)\\
t_{2,5},\ p_2 &: \left(\ell_7^{p_2},\ \textsf{true} \to (trn,\ c_2) := (1,\ 0),\ \ell_0^{p_2}\right)\\
t_{2,6},\ p_2 &: \left(\ell_2^{p_2},\ c_1 = 1 \to (\,) := (\,),\ \ell_3^{p_2}\right)\\
t_{2,7},\ p_2 &: \left(\ell_3^{p_2},\ trn = 2 \to (\,) := (\,),\ \ell_2^{p_2}\right)\\
t_{2,8},\ p_2 &: \left(\ell_3^{p_2},\ trn \neq 2 \to (\,) := (\,),\ \ell_4^{p_2}\right)\\
t_{2,9},\ p_2 &: \left(\ell_4^{p_2},\ \textsf{true} \to (c_2) := (0),\ \ell_5^{p_2}\right)\\
t_{2,10},\ p_2 &: \left(\ell_5^{p_2},\ trn = 1 \to (\,) := (\,),\ \ell_5^{p_2}\right)\\
t_{2,11},\ p_2 &: \left(\ell_5^{p_2},\ trn \neq 1 \to (\,) := (\,),\ \ell_1^{p_2}\right)
\end{aligned}
$$

A semantics for FCS

We need to define the rules that describe the possible behaviors of a FCS, in other words to provide a semantics for FCS.

Note: The semantics of a language defines the meaning of what can be expressed in this language. To be useful, it must be expressed in a simpler language.

Our semantics for FCS will be expressed as a transition system (or automaton).

Transition systems

A transition system is defined by a tuple (S, Σ, s_0, T), where

- S is a (finite or infinite) set of states.
- Σ is an alphabet, i.e., a finite set of distinct symbols.
- $s_0 \in S$ is the initial state.
- $T \subseteq S \times \Sigma \times S$ is a set of transitions.

  Each transition is a triple (s_1, σ, s_2) composed of

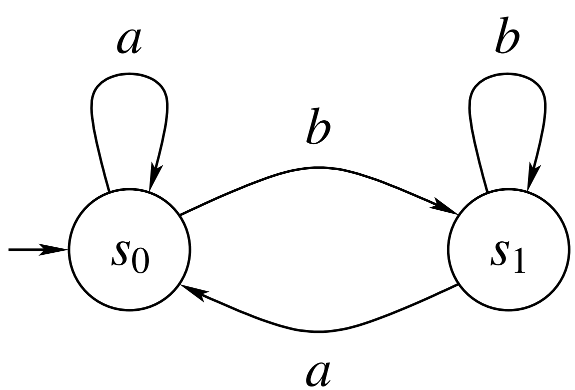
  - a source state s_1.
  - a label σ.
  - a destination state s_2.

Graphical representation

If a transition system has finitely many states, then it can be represented by a finite graph:

- Each state is represented by a node. The initial state is shown with an arrow.
- Each transition is represented by a labeled edge.

Example:



$$
\begin{aligned}
S &= \{s_0, s_1\} \\
\Sigma &= \{a, b\} \\
T &= \{(s_0, a, s_0),\ (s_0, b, s_1),\ (s_1, a, s_0),\ (s_1, b, s_1)\}
\end{aligned}
$$

A semantics for transition systems

A transition system can itself be given a semantics, which takes the form of a set of behaviors.

An infinite behavior ρ of a transition system (S, Σ, s_0, T) is a pair of functions $\rho_S : \mathbb{N} \rightarrow S$ and $\rho_T : \mathbb{N} \rightarrow T$, often written as

$$\rho_S(0) \stackrel{\rho_T(0)}{\rightarrow} \rho_S(1) \stackrel{\rho_T(1)}{\rightarrow} \rho_S(2) \stackrel{\rho_T(2)}{\rightarrow} \cdots$$

such that

- $\rho_S(0) = s_0$, and
- $\forall i \geq 0 : \ (\rho_S(i), \rho_T(i), \rho_S(i+1)) \in T.$

Note: Finite behaviors can be defined similarly, as functions defined on an interval $[0, k]$ instead of $\mathbb{N}$.

The transition system corresponding to a FCS

Problem: Given a FCS $(\mathcal{P}, \mathcal{M}, \mathcal{T})$, how can we compute a transition system (S, Σ, s_0, T) that describes its semantics?

Solution:

- If $\mathcal{P} = \{p_1, \ldots, p_m\}$, $\mathcal{M} = \{x_1, \ldots, x_q\}$, and the domain of each variable x_i is D_i, then the set of states is

$$S = \ell(p_1) \times \cdots \times \ell(p_m) \times D_1 \times \cdots \times D_q.$$

  In other words, each state $s \in S$ takes the form

$$(s(p_1), \ldots, s(p_m), s(x_1), \ldots, s(x_q)),$$

  which associates

  - one control location $s(p_i) \in \ell(p_i)$ to each process p_i, and
  - one value $s(x_i) \in D_i$ to each variable x_i.

- The alphabet is

$$\Sigma = \{n(t) \mid t \in \mathcal{T}\}.$$

- The initial state is

$$s_0 = \left(\ell_0^{p_1}, \ldots, \ell_0^{p_m}, \mathcal{I}(x_1), \ldots, \mathcal{I}(x_q)\right).$$

(Recall that $\mathcal{I}$ is the function that assigns the initial value of variables.)

- The set T of transitions is computed as follows:

  For each transition $t \in \mathcal{T}$ and state $s \in S$, if

  - for each $p_i \in p(t)$, i.e., each process that is active for t, we have $s(p_i) = \ell_s(t, p_i)$, and
  - the condition $C(t)$ is satisfied by the values $(s(x_1), \ldots, s(x_q))$ of the variables in s,

  then the transition t is said to be enabled in s.

If t is enabled in s, then the transition system contains a transition

$$(s, n(t), s'),$$

where s' is the state defined as follows:

- For each $p_i \in p(t)$, i.e., process that is active for t, we have $s'(p_i) = \ell_d(t, p_i)$.
- For each $p_i \notin p(t)$, i.e., process that is not active for t, we have $s'(p_i) = s(p_i)$.
- For each variable x_i to which $A(t)$ assigns an expression exp_i, we have $s'(x_i) = [exp_i]_s$, where $[exp_i]_s$ denotes the result produced by the evaluation of exp_i in which the variables are replaced by their value in s.
- For each variable x_i that is not modified by $A(t)$, we have $s'(x_i) = s(x_i)$.

Note: With this definition, for a given state s and transition $t \in \mathcal{T}$ that is enabled in s, the state s' is unique.

Fairness constraints

The semantics that we have introduced for FCS is an interleaving semantics: The behaviors of the transition system are obtained by interleaving those of the processes.

Problem: This semantics allows some behaviors that are not realistic, in particular those in which some process is deliberately ignored even though it could progress.

Example: Our model of Dekker's algorithm admits the behavior

$$
\begin{aligned}
\left(\ell_0^{p_1}, \ell_0^{p_2}, 0, 0, 1\right) \stackrel{rem_1}{\rightarrow} \left(\ell_1^{p_1}, \ell_0^{p_2}, 0, 0, 1\right) &\stackrel{rem_2}{\rightarrow} \left(\ell_1^{p_1}, \ell_1^{p_2}, 0, 0, 1\right) \\
\stackrel{t_{1,2}}{\rightarrow} \left(\ell_2^{p_1}, \ell_1^{p_2}, 1, 0, 1\right) &\stackrel{t_{2,2}}{\rightarrow} \left(\ell_2^{p_1}, \ell_2^{p_2}, 1, 1, 1\right) \\
\stackrel{t_{1,6}}{\rightarrow} \left(\ell_3^{p_1}, \ell_2^{p_2}, 1, 1, 1\right) &\stackrel{t_{1,7}}{\rightarrow} \left(\ell_2^{p_1}, \ell_2^{p_2}, 1, 1, 1\right) \\
\stackrel{t_{1,6}}{\rightarrow} \left(\ell_3^{p_1}, \ell_2^{p_2}, 1, 1, 1\right) &\stackrel{t_{1,7}}{\rightarrow} \left(\ell_2^{p_1}, \ell_2^{p_2}, 1, 1, 1\right) \\
\stackrel{t_{1,6}}{\rightarrow} \quad \ldots &
\end{aligned}
$$

in which p_2 is stalled forever.

A consequence is that it would be impossible to prove properties that require the progress of one or many processes.

Solution: Impose fairness assumptions on behaviors.

Principles:

- Fairness assumptions are qualitiative, not quantitative: They can express that a process cannot wait indefinitely before performing some action, but not for how long it can wait.
- Fairness assumptions can be seen as abstractions of concrete process scheduling policies, such as those implemented by operating systems.

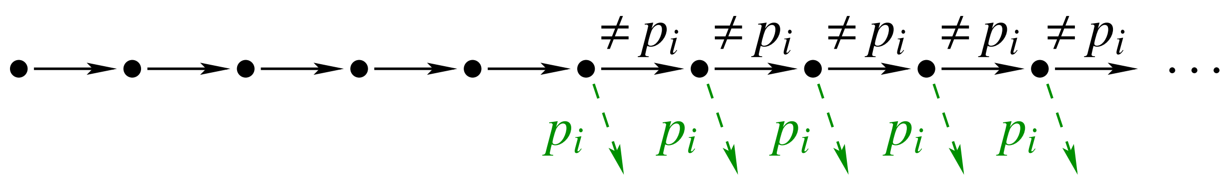
  If a fairness assumption is used for verifying a property of a system, it guarantees that this property holds for any implementation that satisfies this assumption.
- There exist several forms of fairness assumptions. One of the least restrictive is called weak fairness.

Weak fairness

We define weak fairness at the level of processes. (There exists another notion of weak fairness based on transitions.)

It is easier to first define the negation of weak fairness:

Definition: A behavior is not weakly fair if, for some process p_i and some point on, there always exists an enabled transition in which p_i is active, and no such transition is taken.



We then have the following positive definition:

Definition: A behavior is weakly fair if, for every process p_i, it is the case that infinitely often

- either no transition in which p_i is active is enabled, or
- a transition in which p_i is active is followed.

In other words, weak fairness expresses that every process that needs to progress must, from some point on, always get a chance to do so, and must eventually take this chance.

Observer processes

Question: How do we impose fairness assumptions on the system's behaviors?

Answer: By adding observer processes.

Principles:

- An observer process is a process that does not affect the behaviors of a FCS, but observes them in order to check some properties.
- An observer process is similar to a regular process, except that its transitions
  - do not contain assignments, and
  - admit a more general form of conditions.

Formally, an observer process ob for a FCS $(\mathcal{P}, \mathcal{M}, \mathcal{T})$ is defined by the following elements:

- A finite set $\ell(ob)$ of control locations.
- An initial location $\ell_0^{ob} \in \ell(ob)$.
- A finite set $\mathcal{T}_{ob}$ of transitions.

A transition $\tau \in \mathcal{T}_{ob}$ of an observer process ob is characterized by the following elements:

- A source location $\ell_s(\tau) \in \ell(ob)$.
- A destination location $\ell_d(\tau) \in \ell(ob)$.
- A Boolean condition $C(\tau)$ expressed in terms of
  - a location variable loc_{p_i} for each (non-observer) process $p_i \in \mathcal{P}$.
  - the variables in $\mathcal{M}$.
  - a variable tr whose value is an element of the set $\mathcal{T}$ of transitions of the FCS.

Notation:

- Such a transition τ will usually be written $(\ell_s(\tau), C(\tau), \ell_d(\tau))$.
- We will write

$$C(\tau)[loc_{p_1}, \ldots, loc_{p_m}, x_1, \ldots, x_q, tr]$$

  when we wish to make explicit the variables involved in $C(\tau)$.

The conditions in observer processes

The conditions in the transitions of observer processes take the form of Boolean formulas built from atomic formulas that belong to the following (non exhaustive) list:

- Computable predicates applied to the variables in $\mathcal{M}$.

  Examples: $x_1 \leq x_2, \; x_3 = 10$.

- Equalities and inequalities between a location variable for a process and a control location of that process.

  Examples: $loc_{p_2} = \ell_3^{p_2}, \; loc_{p_1} \neq \ell_0^{p_1}$.

- Comparisons between the components of the transition tr and the elements of transitions in the FCS.

  Examples: $n(tr) = t_{1,2}, \; p_1 \in p(tr), \; \ell_s(tr, p_2) = \ell_4^{p_2}$.

Example: Dekker's algorithm

Exercise: Write an observer process for our model of Dekker's algorithm, that checks whether both processes can enter simultaneously their critical section.

Solution: Detect whether a process can follow its $crit$ transition when the other process is in the ℓ_6 location.

- The set of locations is $\ell(ob) = \left\{\ell_0^{ob}, \ell_1^{ob}\right\}$, with ℓ_0^{ob} initial.
- The only transition is

$$\left(\ell_0^{ob}, loc_{p_1} = \ell_6^{p_1} \wedge loc_{p_2} = \ell_6^{p_2}, \ell_1^{ob}\right).$$

The problem of checking that mutual exclusion is not always satisfied then amounts to checking whether the process ob can reach the control location ℓ_1^{ob}.

Composing observers with a FCS

When building the transition system corresponding to FCS, observer processes should not be handled in the same way as other processes: Their transitions must not be interleaved with those of the other processes, but synchronized with them.

In order words, each time that the FCS follows a transition, the observers must also follow a transition. If an observer does not have an enabled transition, we consider that it follows a default transition that keeps it in the same state.

Let $(\mathcal{P}, \mathcal{M}, \mathcal{T})$ be a formal concurrent system, and $ob_1 = (\ell(ob_1), \ell_0^{ob_1}, \mathcal{T}_{ob_1})$, $ob_2 = (\ell(ob_2), \ell_0^{ob_2}, \mathcal{T}_{ob_2}), \ldots, ob_k = (\ell(ob_k), \ell_0^{ob_k}, \mathcal{T}_{ob_k})$ be observer processes. The corresponding transition system (S, Σ, s_0, T) is defined as follows.

- If $\mathcal{P} = \{p_1, \ldots, p_m\}$, $\mathcal{M} = \{x_1, \ldots, x_q\}$, and the domain of each variable x_i is D_i, then the set of states is

$$S = \ell(p_1) \times \cdots \times \ell(p_m) \times \ell(ob_1) \times \cdots \times \ell(ob_k) \times D_1 \times \cdots \times D_q.$$

  Note: In the same way as regular processes, for $s \in S$, $s(ob_i)$ denotes the location of the observer process ob_i in s.

- The alphabet is

$$\Sigma = \{n(t) \mid t \in \mathcal{T}\}.$$

- The initial state is

$$s_0 = \left(\ell_0^{p_1}, \ldots, \ell_0^{p_m}, \ell_0^{ob_1}, \ldots, \ell_0^{ob_k}, \mathcal{I}(x_1), \ldots, \mathcal{I}(x_q)\right).$$

- The set T of transitions is computed as follows:

  For each transition $t \in \mathcal{T}$ and state $s \in S$, if

  – for each $p_i \in p(t)$, i.e., each process that is active for t, we have $s(p_i) = \ell_s(t, p_i)$, and

  – the condition $C(t)$ is satisfied by the values $(s(x_1), \ldots, s(x_q))$ of the variables in s,

  then the transition t is said to be enabled in s.

  Note: The criterion for a transition to be enabled is not affected by observer processes.

Let t be a transition that is enabled in s. For each observer process ob_i and transition τ_i of this process such that either

- $\tau_i \in \mathcal{T}_{ob_i}$, $\ell_s(\tau_i) = s(ob_i)$, and $C(\tau_i)[s(p_1), \ldots, s(p_m), s(x_1), \ldots, s(x_q), t] =$ true, where $\mathcal{P} = \{p_1, \ldots, p_m\}$ and $\mathcal{M} = \{x_1, \ldots x_q\}$ (if such a transition exists), or
- τ_i is a default transition $(s_{ob_i}, \text{true}, s_{ob_i})$ of ob_i (otherwise),

the transition system contains a transition

$$(s, n(t), s'),$$

where s' is the state defined as follows:

- For each $p_j \in p(t)$, i.e., regular process that is active for t, we have $s'(p_j) = \ell_d(t, p_j)$.
- For each $p_j \notin p(t)$, i.e., regular process that is not active for t, we have $s'(p_j) = s(p_j)$.
- For each observer process ob_i, we have $s'(ob_i) = \ell_d(\tau_i)$.

– For each variable x_j to which $A(t)$ assigns an expression exp_j, we have $s'(x_j) = [exp_j]_s$.

– For each variable x_j that is not modified by $A(t)$, we have $s'(x_j) = s(x_j)$.

Notes: If the model contains observer processes, then for a given state s and transition $t \in \mathcal{T}$ that is enabled in s, the state s' is no longer necessarily unique.

Modeling fairness conditions with observers

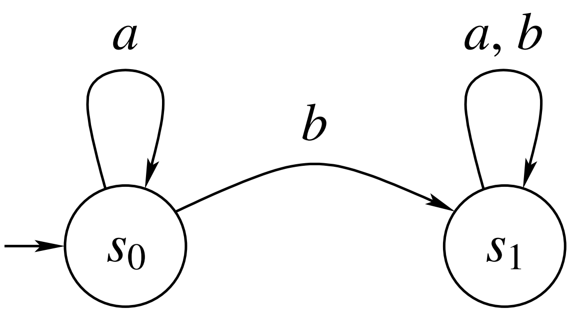
A fairness assumption restricts the possible infinite behaviors of a FCS. It can be modeled by an observer process that distinguishes between fair and unfair behaviors.

This requires to extend our definitions of observer processes and transition systems with a notion of acceptance condition.

Definitions: Let ρ be an infinite behavior of a transition system (S, Σ, s_0, T).

- A state $s \in S$ appears infinitely often in ρ if there exist infinitely many $i \in \mathbb{N}$ such that $\rho_S(i) = s$.
- The set of states in S that appear infinitely often in ρ is denoted by $\mathit{inf}(\rho)$.
- A generalized Büchi acceptance condition for the infinite behaviors of (S, Σ, s_0, T) is a set $F = \{F_1, F_2, \ldots, F_k\}$ of subsets of S.
- The behavior ρ satisfies the acceptance condition $F = \{F_1, F_2, \ldots, F_k\}$ (in other words, ρ is accepted) if for all $F_i \in F$, one has $\mathit{inf}(\rho) \cap F_i \neq \emptyset$.

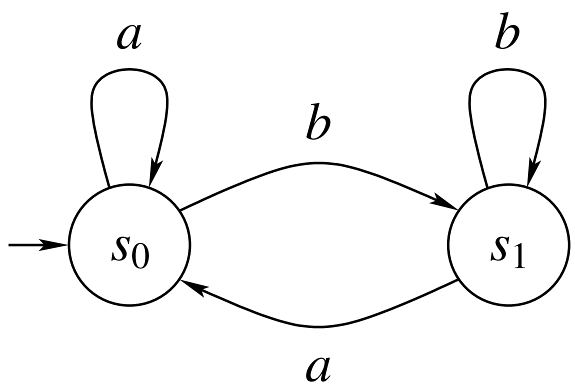
Example 1



$$F = \{\{s_1\}\}$$

The accepted behaviors are those in which a transition labeled by b is followed at least once.

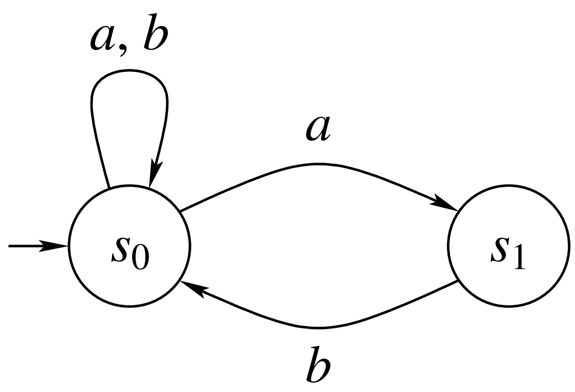
Example 2



$$F = \{\{s_0\}, \{s_1\}\}$$

The accepted behaviors are those in which transitions labeled by a and b are both followed infinitely many times.

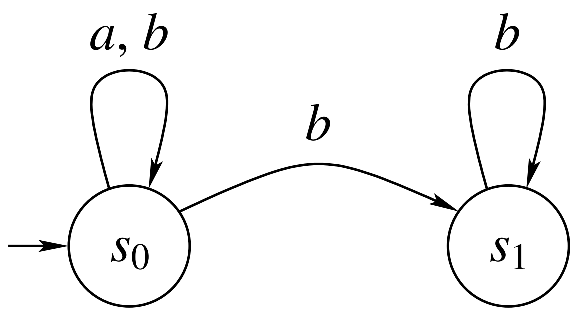
Example 3



$$F = \{\{s_1\}\}$$

The accepted behaviors are those in which infinitely many times, one follows a transition labeled by a and then immediately a transition labeled by b.

Example 4



$$F = \{\{s_1\}\}$$

The accepted behaviors are those that follow finitely many transitions labeled by a and infinitely many labeled by b.

Acceptance conditions for observers

We have defined the notion of acceptance condition with respect to a transition system. A similar definition can be given for observer processes.

Definition: Consider a FCS for which the observer processes are $ob_1, ob_2, \ldots, ob_k$. An acceptance condition for one of these observer processes $ob_i = (\ell(ob_i), \ell_0^{ob_i}, \mathcal{T}_{ob_i})$ is a set

$$F^{ob_i} = \{F_1^{ob_i}, F_2^{ob_i}, \ldots, F_{h_i}^{ob_i}\}$$

where each $F_j^{ob_i}$ is a subset of $\ell(ob_i)$.

The transition system (S, Σ, s_0, T) corresponding to this FCS then admits the accepting condition

$$F = \{F_{1,1}, F_{1,2}, \ldots, F_{1,h_1},\ F_{2,1}, F_{2,2}, \ldots, F_{2,h_2},\ \ldots,\ F_{k,1}, F_{k,2}, \ldots, F_{k,h_k}\},$$

where for each $i \in [1, k]$ and $j \in [1, h_i]$, we have

$$F_{i,j} = \{s \in S \mid s(ob_i) \in F_j^{ob_i}\}.$$

Observers for weak fairness

To impose the weak fairness assumption, we add an observer process for each (regular) process p_i of the FCS.

Those observer processes are described in terms of the two following predicates, evaluated with respect to a state s of the global transition system and a transition t of the FCS:

- $e(p_i)[s, t]$ is true if the process p_i is active for at least one transition that is enabled in s.

  (In other words, $e(p_i)[s, t]$ is true if p_i can progress from s.)

- $a(p_i)[s, t]$ is true if the process p_i is active for the transition t.

For each process p_i of the FCS, one then defines its corresponding observer ob_i as follows:

- Its set of control locations is $\ell(ob_i) = \left\{\ell_0^{ob_i}, \ell_1^{ob_i}\right\}$.

- Its initial location is $\ell_0^{ob_i}$.

- Its set of transitions is $\mathcal{T}_{ob_i} = \left\{ \left(\ell_0^{ob_i},\ e(p_i) \wedge \neg a(p_i),\ \ell_1^{ob_i}\right), \left(\ell_1^{ob_i},\ \neg e(p_i) \vee a(p_i),\ \ell_0^{ob_i}\right) \right\}$.

- Its acceptance condition is $F^{ob_i} = \left\{\left\{\ell_0^{ob_i}\right\}\right\}$.

Notes:

- Such an observer accepts only the infinite behaviors of p_i in which infinitely often either $e(p_i)$ is false, or $a(p_i)$ is true.

- The transitions $\left(\ell_0^{ob_i},\ \neg e(p_i) \vee a(p_i),\ \ell_0^{ob_i}\right)$ and $\left(\ell_1^{ob_i},\ e(p_i) \wedge \neg a(p_i),\ \ell_1^{ob_i}\right)$ are implicit and can be omitted.

Modeling semaphores

Definition: A semaphore is an integer variable s that admits two atomic operations:

- $wait(s)$: if $s > 0$ then $s := s - 1$
  else *suspend until* $s > 0$

- $signal(s)$: $s := s + 1$

Question: How can a semaphore be represented in a FCS?

Answer: A possible solution is to

- represent the value of the semaphore by an integer variable s.

- model a *wait* operation of a process p_i by a transition t of the form

$$t,\ p_i : \left(\ell_1^{p_i},\ s > 0 \ \to\ (s) := (s - 1),\ \ell_2^{p_i}\right),$$

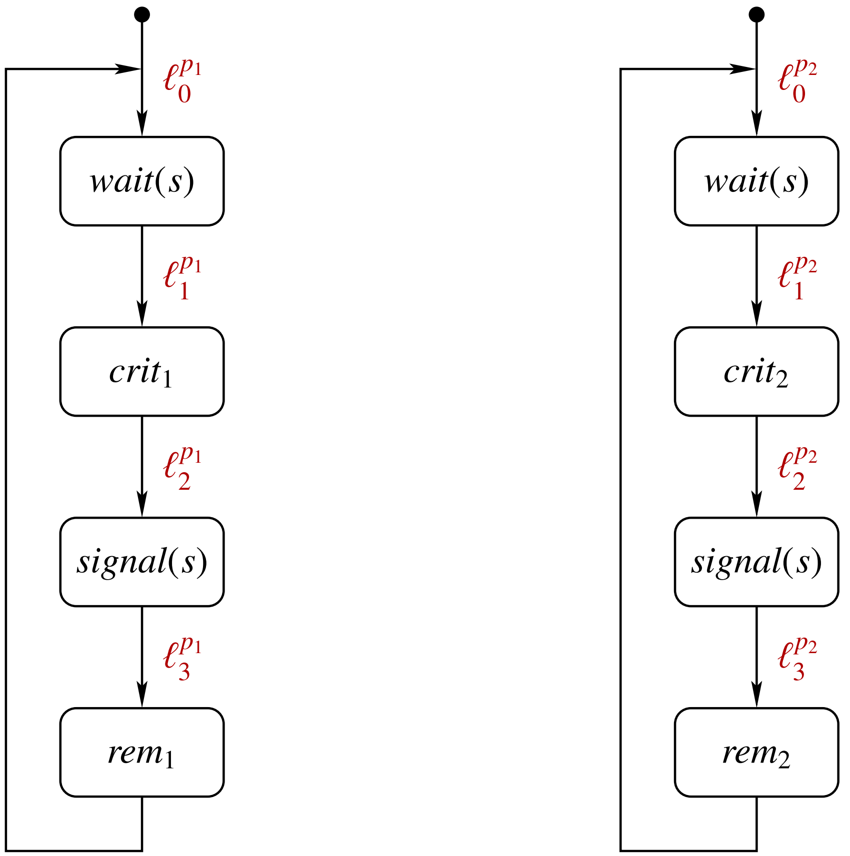
where $\ell_1^{p_i}$ and $\ell_2^{p_i}$ are respectively the source and destination locations of the operation.

- model a *signal* operation of a process p_i by a transition t of the form

$$t,\ p_i : \left(\ell_1^{p_i},\ \text{true}\ \rightarrow\ (s)\ :=\ (s+1),\ \ell_2^{p_i}\right),$$

Problem: With this model, weak fairness assumptions are not sufficient for preventing process starvation.

Illustration: (initially, $s = 1$)



This FCS admits the behavior

$$
\begin{aligned}
\left(\ell_0^{p_1}, \ell_0^{p_2}, 1\right) &\overset{wait_1}{\to} \left(\ell_1^{p_1}, \ell_0^{p_2}, 0\right) \overset{crit_1}{\to} \left(\ell_2^{p_1}, \ell_0^{p_2}, 0\right) \\
&\overset{signal_1}{\to} \left(\ell_3^{p_1}, \ell_0^{p_2}, 1\right) \overset{rem_1}{\to} \left(\ell_0^{p_1}, \ell_0^{p_2}, 1\right) \\
&\overset{wait_1}{\to} \left(\ell_1^{p_1}, \ell_0^{p_2}, 0\right) \overset{crit_1}{\to} \left(\ell_2^{p_1}, \ell_0^{p_2}, 0\right) \\
&\overset{signal_1}{\to} \left(\ell_3^{p_1}, \ell_0^{p_2}, 1\right) \overset{rem_1}{\to} \left(\ell_0^{p_1}, \ell_0^{p_2}, 1\right) \\
&\overset{wait_1}{\to} \quad \ldots
\end{aligned}
$$

in which the process p_2 never progresses, even though this behavior satisfies the weak fairness assumption.

Indeed, in this behavior:

- $a(p_1)$ is infinitely often true (actually, it is always true), and
- $e(p_2)$ is infinitely often false.

Modeling fairness for semaphores

Weak fairness assumptions are thus not sufficient for accurately describing the behavior of semaphores. There are two approaches to solving this problem.

Solution 1: Represent explicitly the process scheduling policy.

Illustration: (FIFO semaphores)

- The process table is represented by an array variable $status$ that associates to each process p_i the symbol
  - R if p_i is running, and
  - W if p_i is waiting on a semaphore.
- The wait queue of a semaphore s is represented by a variable $list_s$ that contains a FIFO list of the processes waiting for s.

- In the process p_i, the *wait*(s) operation is modeled by three transitions of the form

$$\begin{array}{lcl} t_1, p_i & : & \left(\ell_1^{p_i},\ s > 0 \ \rightarrow\ (s)\ :=\ (s-1),\ \ell_2^{p_i}\right) \\ t_2, p_i & : & \left(\ell_1^{p_i},\ s \leq 0 \ \rightarrow\ (status[p_i],\ list_s)\ :=\ (W,\ list_s \cdot p_i),\ \ell_3^{p_i}\right) \\ t_3, p_i & : & \left(\ell_3^{p_i},\ status[p_i] = R \ \rightarrow\ (\,)\ :=\ (\,),\ \ell_2^{p_i}\right), \end{array}$$

  where $\ell_1^{p_i}$ and $\ell_2^{p_i}$ are respectively the source and destination control locations of the operation, and $\ell_3^{p_i}$ is an auxiliary location.

- The *signal*(s) operation is modeled by

$$\begin{array}{lcl} t_1, p_i & : & \left(\ell_1^{p_i},\ list_s = [\,] \ \rightarrow\ (s)\ :=\ (s+1),\ \ell_2^{p_i}\right) \\ t_2, p_i & : & \left(\ell_1^{p_i},\ list_s \neq [\,] \ \rightarrow\ (status[head(list_s)],\ list_s)\ :=\ (R,\ tail(list_s)),\ \ell_2^{p_i}\right), \end{array}$$

  where $\ell_1^{p_i}$ and $\ell_2^{p_i}$ are respectively the source and destination control locations.

Drawbacks:

- This approach is only valid for a particular process scheduling policy.
- The size of the corresponding transition system is increased.

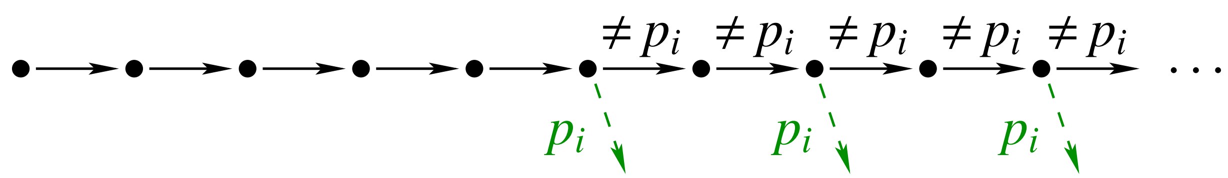
Solution 2: Introduce a stronger form of fairness assumptions.

Strong fairness

Like for weak fairness, we define strong fairness at the level of processes. (There exists another notion based on transitions.)

The negation of strong fairness is defined as follows.

Definition: A behavior is not strongly fair if for some process p_i, there is infinitely often an enabled transition for which p_i is active, but from some point on, no such transition is taken.



A positive definition is then:

Definition: A behavior is strongly fair if, for every process p_i,

- either from some point on, no transition for which p_i is active is enabled, or
- a transition in which p_i is active is followed infinitely often.

In other words, strong fairness expresses that every process that needs to progress must, from some point on, get infinitely often a chance to do so, and must eventually take this chance.

Advantage: With strong fairness assumptions, the simple model of semaphores is sufficient for proving most properties of interest.

Observers for strong fairness

The weak fairness assumption can be imposed by adding for each (regular) process p_i of the FCS an observer process ob_i defined as follows.

- Its set of control locations is $\ell(ob_i) = \left\{\ell_0^{ob_i}, \ell_1^{ob_i}, \ell_2^{ob_i}, \ell_3^{ob_i}\right\}$.

- Its initial location is $\ell_0^{ob_i}$.

- Its set of transitions is $$\mathcal{T}_{ob_i} = \left\{ \left(\ell_0^{ob_i},\ \neg e(p_i),\ \ell_0^{ob_i}\right), \left(\ell_0^{ob_i},\ a(p_i),\ \ell_1^{ob_i}\right), \left(\ell_1^{ob_i},\ \text{true},\ \ell_0^{ob_i}\right), \left(\ell_0^{ob_i},\ \neg e(p_i),\ \ell_2^{ob_i}\right), \left(\ell_2^{ob_i},\ e(p_i),\ \ell_3^{ob_i}\right) \right\}.$$

- Its acceptance condition is $F^{ob_i} = \left\{\left\{\ell_1^{ob_i},\ \ell_2^{ob_i}\right\}\right\}$.

Notes:

- Those observers are nondeterministic.
- The observer for p_i accepts only the infinite behaviors in which
  - either $e(p_i)$ is, from some point on, always false, or
  - $a(p_i)$ is true infinitely often.

Chapter 3

Infinite-word automata

Words and automata

Finite case:

- A finite word w of length n over an alphabet Σ is a mapping

$$w : \{0, 1, \ldots, n-1\} \rightarrow \Sigma.$$

- An automaton on finite words is a finite-state machine that accepts a set (language) of finite words.

Infinite case:

- An infinite word w over an alphabet Σ is a mapping

$$w : \mathbb{N} \rightarrow \Sigma.$$

- An automaton on infinite words is a finite-state machine that accepts a set (language) of infinite words.

Note: Automata are sometimes introduced as models of computation. Here, we just view them as descriptions of languages.

Automata on infinite words

Definition: A Büchi automaton is a tuple $(S, \Sigma, \delta, S_0, F)$, where

- S is a finite set of states.
- Σ is a finite alphabet.
- δ is a transition function, either of the form
  - $\delta : S \times \Sigma \to S$ (deterministic automaton), or
  - $\delta : S \times \Sigma \to 2^S$ (nondeterministic automaton).
- $S_0 \subseteq S$ is a set of initial states.
- $F \subseteq S$ is a set of accepting states.

Note: This structure is similar to the one of finite-word automata. The difference is in their semantics.

Semantics of finite-word automata

Note: We consider the more general case of nondeterministic automata.

Semantics: A word

$$w : \{0, 1, \ldots, n-1\} \to \Sigma$$

is accepted by an automaton $(S, \Sigma, \delta, S_0, F)$ if there exists a labeling

$$\lambda : \{0, 1, \ldots n\} \to S$$

such that

- $\lambda(0) \in S_0$ (the initial label is an initial state).
- $\forall i \in [0, n-1] : \lambda(i+1) \in \delta(w(i), \lambda(i))$ (the labeling is compatible with the transition relation).
- $\lambda(n) \in F$ (the last label is an accepting state).

Semantics of Büchi automata

A word

$$w : \mathbb{N} \to \Sigma$$

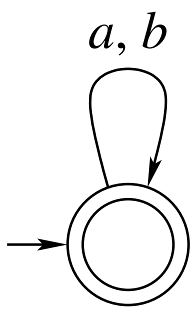
is accepted by an automaton $(S, \Sigma, \delta, S_0, F)$ if there exists a labeling

$$\lambda : \mathbb{N} \to S$$

such that

- $\lambda(0) \in S_0$ (the initial label is an initial state).
- $\forall i \in \mathbb{N} : \lambda(i+1) \in \delta(w(i), \lambda(i))$ (the labeling is compatible with the transition relation).
- $inf(\lambda) \cap F \neq \emptyset$ (the set of repeating states intersects F).

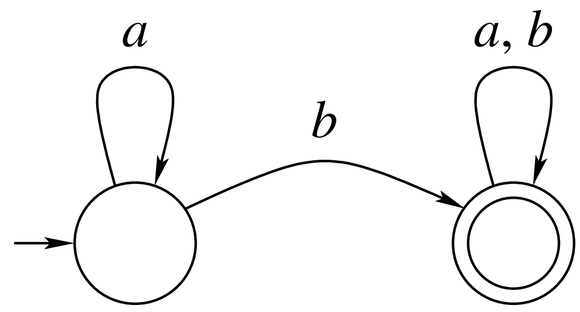
Example 1



This automaton accepts all the infinite words over $\Sigma = \{a, b\}$.

Note: The accepting states are denoted by double circles.

Example 2

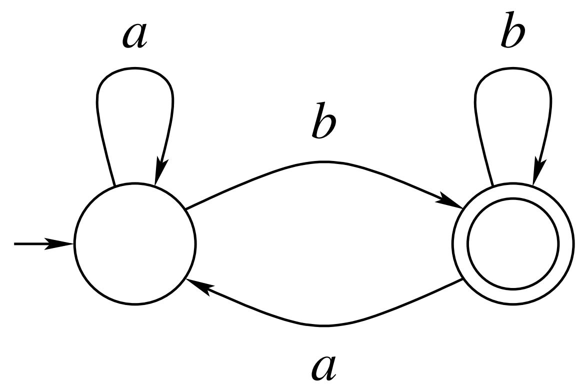


This automaton accepts all the infinite words composed of

- a finite number of symbols a, followed by
- one copy of b, followed by
- any infinite suffix over $\Sigma = \{a, b\}$.

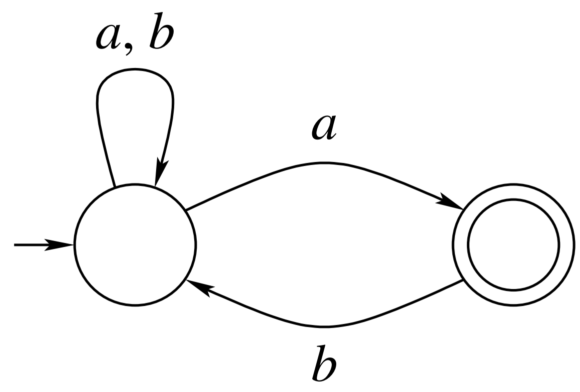
Note: Such a language can de described by an ω-regular expression $a^* b\,(a + b)^\omega$, where * and $^\omega$ denote respectively finite and infinite repetition.

Example 3



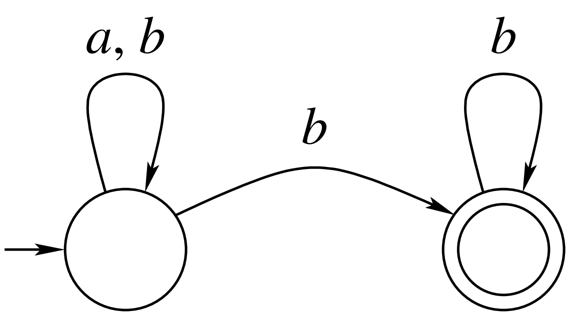
This automaton accepts all the infinite words over $\Sigma = \{a, b\}$ that contain infinitely many symbols b.

Example 4



This automaton accepts all the infinite words over $\Sigma = \{a, b\}$ that contain infinitely many symbols a immediately followed by a symbol b.

Example 5



This automaton accepts all the infinite words over $\Sigma = \{a, b\}$ that contain finitely many symbols a.

Different types of acceptance conditions

- Büchi: Accepting states: $F \subseteq S$. Condition on labelings: $inf(\lambda) \cap F \neq \emptyset$.
- Generalized Büchi: Accepting states: $F \subseteq 2^S$, i.e., $F = \{F_1, \ldots, F_k\}$ with $\forall i : F_i \subseteq S$. Condition on labelings: For each F_i, $inf(\lambda) \cap F_i \neq \emptyset$.
- Rabin: Accepting states: $F \subseteq 2^S \times 2^S$, i.e., $F = \{(G_1, B_1), (G_2, B_2), \ldots, (G_k, B_k)\}$. Condition on labelings: For some pair $(G_i, B_i) \in F$, $inf(\lambda) \cap G_i \neq \emptyset$ and $inf(\lambda) \cap B_i = \emptyset$.
- Streett: Accepting states: $F \subseteq 2^S \times 2^S$, i.e., $F = \{(G_1, B_1), (G_2, B_2), \ldots, (G_k, B_k)\}$. Condition on labelings: For all pairs $(G_i, B_i) \in F$, $inf(\lambda) \cap G_i \neq \emptyset$ or $inf(\lambda) \cap B_i = \emptyset$.
- Muller: Accepting states: $F \subseteq 2^S$, i.e., $F = \{F_1, \ldots, F_k\}$ with $\forall i : F_i \subseteq S$. Condition on labelings: For some F_i, $inf(\lambda) = F_i$.

The expressiveness of infinite-word automata

All those forms of infinite-word automata (in their nondeterministic version) share the same expressive power: They all define the ω-regular languages

$$\bigcup_i L_{1,i} \, L_{2,i}^{\omega},$$

where the union is finite, each $L_{1,i}$ and $L_{2,i}$ is a regular finite-word language, and $^{\omega}$ denotes infinite repetition.

For deterministic automata, Büchi and Generalized Büchi are weaker: There exist ω-regular languages that cannot be accepted by such automata.

Example: $(a + b)^* \, b^{\omega}$.

In contrast, the deterministic and nondeterministic forms of Rabin, Streett and Muller automata share the same expressive power, which is identical to that of nondeterministic Büchi automata.

Operations on infinite-word automata

Nondeterministic Büchi:

- These automata are closed under union, intersection, projection, and complementation.

  Notes:

  - Projection is an operation defined by a morphism $\pi : \Sigma_1 \to \Sigma_2$ that maps every symbol of the alphabet Σ_1 onto a symbol in another alphabet Σ_2.
  - Complementation of Büchi automata is a difficult operation.

- Nonemptiness can be decided in linear time (by computing the reachable strongly connected components).

Rabin: Nonemptiness can be decided in polynomial time (by a conversion to nondeterministic Büchi automata).

Street & Muller: The conversion to nondeterministic Büchi automata incurs an exponential blowup.

Union of Büchi automata

Problem: Given two Büchi automata $\mathcal{A}_1 = (S_1, \Sigma, \delta_1, S_{01}, F_1)$ and $\mathcal{A}_2 = (S_2, \Sigma, \delta_2, S_{02}, F_2)$, compute an automaton $\mathcal{A} = (S, \Sigma, \delta, S_0, F)$ that accepts the union of the languages accepted by $\mathcal{A}_1$ and $\mathcal{A}_2$.

Solution:

- $S = S_1 \cup S_2$ (assuming w.l.o.g. that $S_1 \cap S_2 = \emptyset$).
- $S_0 = S_{01} \cup S_{02}$.
- $s' \in \delta(s, a)$ if either
  - $s' \in \delta_1(s, a)$ and $s \in S_1$, or
  - $s' \in \delta_2(s, a)$ and $s \in S_2$.
- $F = F_1 \cup F_2$.

Intersection of Büchi automata

Problem: Given two Büchi automata $\mathcal{A}_1 = (S_1, \Sigma, \delta_1, S_{01}, F_1)$ and $\mathcal{A}_2 = (S_2, \Sigma, \delta_2, S_{02}, F_2)$, compute an automaton $\mathcal{A} = (S, \Sigma, \delta, S_0, F)$ that accepts the intersection of the languages accepted by $\mathcal{A}_1$ and $\mathcal{A}_2$.

Solution: We first construct a generalized Büchi automaton:

- $S = S_1 \times S_2$.
- $S_0 = S_{01} \times S_{02}$.
- $(s_1', s_2') \in \delta((s_1, s_2), a)$ if both
  - $s_1' \in \delta_1(s_1, a)$ and
  - $s_2' \in \delta_2(s_2, a)$.
- $F = \{F_1 \times S_2,\ S_1 \times F_2\}$.

From generalized Büchi to Büchi

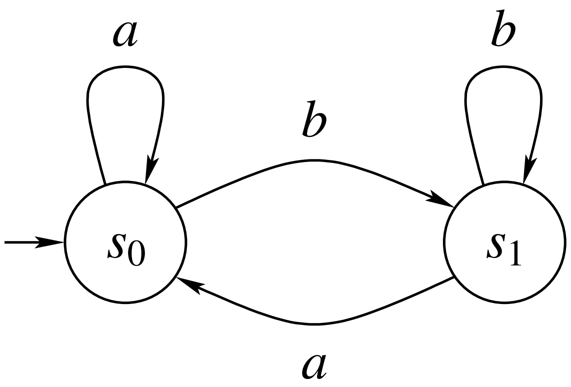
Problem: Given a generalized Büchi automaton $\mathcal{A} = (S, \Sigma, \delta, S_0, F)$, compute a Büchi automaton $\mathcal{A}' = (S', \Sigma, \delta', S'_0, F')$ that accepts the same language.

Solution: Let $F = \{F_1, F_2, \ldots F_k\}$. We have

- $S' = S \times \{1, 2, \ldots, k\}$.
- $S'_0 = S_0 \times \{1\}$.
- $(s', j) \in \delta'((s, i), a)$ if $s' \in \delta(s, a)$ and either
  - $s \notin F_i$ and $j = i$, or
  - $s \in F_i$ and $j = (i \mod k) + 1$.
- $F' = F_1 \times \{1\}$.

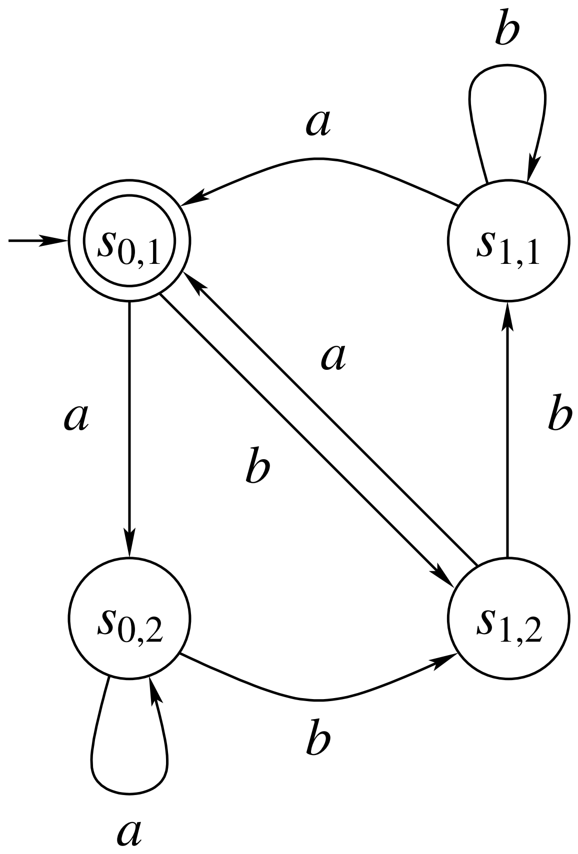
Illustration

Generalized Büchi automaton:



$$F_1 = \{s_0\}, \quad F_2 = \{s_1\}$$

Equivalent Büchi automaton:



Chapter 4

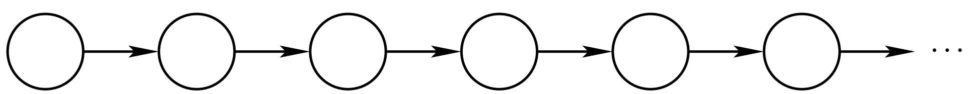
Temporal logic

Introduction

- Temporal logics are formalisms used for expressing properties of infinite sequences.

- They extend propositional or first-order logic.

- In this course, we will study Linear-time Temporal Logic (LTL) on discrete time.
  - A LTL formula is interpreted on states belonging to a sequence:

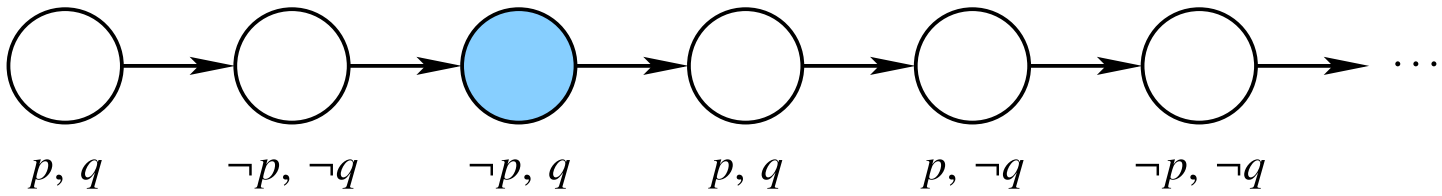


  - Each state assigns a truth value to atomic propositions.
  - Temporal operators indicate in which states the components of formulas should be interpreted.

Temporal operators

- $\bigcirc\,\varphi$ (Next): φ is true in the next state of the sequence.
- $\Box\,\varphi$ (Always): φ is true in the current and all future states of the sequence.
- $\Diamond\,\varphi$ (Eventually): φ is true in the current or some future state of the sequence.
- $\varphi_1\, U\, \varphi_2$ (Until): φ_1 is true in the current and all future states until φ_2 becomes true, which must occur.
- $\varphi_1\, \tilde{U}\, \varphi_2$ (Releases): φ_2 is true in the current and all future states, until this obligation is released by φ_1 being true in a previous state, which will not necessarily occur.

Examples



In the colored state, the following formulas are true:

- $\neg p \wedge q$
- $\bigcirc(p \wedge q)$
- $\Diamond \neg q$

In that state, the following formulas are false:

- $\Box q$
- $\bigcirc \Box p$

More examples

- $\Box(p \Rightarrow \bigcirc q)$ is satisfied in all states where, for this state and all future ones, each state in which p is true is immediately followed by a state in which q is true.
- $\Box(p \Rightarrow \bigcirc(\neg q \, U \, r))$ is satisfied in all states where, for this state and all future ones, after each state in which p is true, q is false from the next state on, and remains false until the first state where r is true, which must occur.
- $\Box \Diamond p$ is satisfied in all states for which p is true infinitely often in the future.
- $\Diamond \Box p$ is satisfied in all states for which, from some point on in the future, p becomes and remains true.

Syntax of LTL

The formulas of LTL built from a set P of atomic propositions are the following:

- true, false, as well as p and $\neg p$ for all $p \in P$.
- $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$, where φ_1 and φ_2 are LTL formulas.
- $\bigcirc \varphi_1$, $\varphi_1 \, U \, \varphi_2$, and $\varphi_1 \, \tilde{U} \, \varphi_2$, where φ_1 and φ_2 are LTL formulas.

The operators $\Diamond$ and $\Box$ are then defined as the following abbreviations:

- $\Diamond \varphi \;:=\;$ true $U \, \varphi$
- $\Box \varphi \;:=\;$ false $\tilde{U} \, \varphi$

Semantics of LTL

The semantics of LTL is defined with respect to paths $\pi : \mathbb{N} \to 2^P$ (in other words, infinite sequences of subsets of atomic propositions).

Notation: For a path π and $i \geq 0$, the notation π^i represents the suffix of π obtained by removing its i first states, in other words, $\pi^i(j) = \pi(i + j)$ for all $j \geq 0$.

Semantic rules: The following rules give the truth value of a formula in the first state of a path π:

- $\pi \models$ true and $\pi \not\models$ false (for any path π).
- For each $p \in P$:
  - $\pi \models p$ iff $p \in \pi(0)$.
  - $\pi \models \neg p$ iff $p \notin \pi(0)$.
- $\pi \models \varphi_1 \wedge \varphi_2$ if $\pi \models \varphi_1$ and $\pi \models \varphi_2$.
- $\pi \models \varphi_1 \vee \varphi_2$ if $\pi \models \varphi_1$ or $\pi \models \varphi_2$.

- $\pi \models \bigcirc\varphi$ iff $\pi^1 \models \varphi$.
- $\pi \models \varphi_1\, U\, \varphi_2$ iff there exists $i \geq 0$ such that $\pi^i \models \varphi_2$ and for all j such that $0 \leq j < i$, we have $\pi^j \models \varphi_1$.
- $\pi \models \varphi_1\, \tilde{U}\, \varphi_2$ iff for all $i \geq 0$ such that $\pi^i \not\models \varphi_2$, there exists j such that $0 \leq j < i$ and $\pi^j \models \varphi_1$.

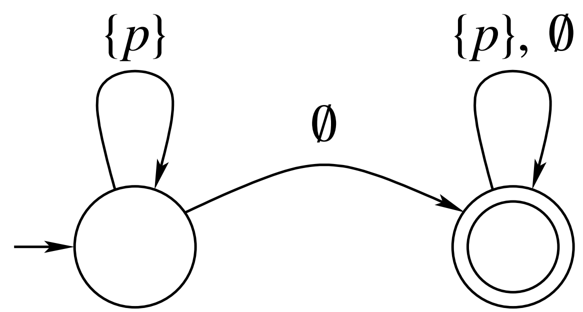
Notes:

- As we have defined it, the syntax of the logic only allows to apply negation to atomic propositions. This restriction can be lifted with the help of additional definitions:
  - $\pi \not\models \varphi_1\, U\, \varphi_2$ iff $\pi \models (\neg\varphi_1)\, \tilde{U}\, (\neg\varphi_2)$
  - $\pi \not\models \varphi_1\, \tilde{U}\, \varphi_2$ iff $\pi \models (\neg\varphi_1)\, U\, (\neg\varphi_2)$
  - $\pi \not\models \bigcirc\varphi$ iff $\pi \models \bigcirc\neg\varphi$
- A LTL formula can be seen as a description of a set of infinite sequences (those that satisfy it).

From LTL to automata

Problem: Given a LTL formula φ built from a set P of atomic propositions, construct algorithmically an infinite-word automaton over the alphabet 2^P that accepts exactly the infinite sequences satisfying φ.

Example: $\varphi = \Diamond \neg p$, with $P = \{p\}$:



(Büchi acceptance condition)

Closure of a formula

We first look at the problem of determining whether a sequence $\pi : \mathbb{N} \to 2^P$ satisfies a formula φ.

A possible solution is to label the positions of π with subformulas of φ, in a way that is consistent with LTL semantics. The subformulas of φ that need to be considered form the closure $cl(\varphi)$ of φ.

Definition: $cl(\varphi)$ is the smallest set of formulas such that

- $\varphi \in cl(\varphi)$.
- If $\varphi_1 \wedge \varphi_2 \in cl(\varphi)$, then $\varphi_1 \in cl(\varphi)$ and $\varphi_2 \in cl(\varphi)$.
- If $\varphi_1 \vee \varphi_2 \in cl(\varphi)$, then $\varphi_1 \in cl(\varphi)$ and $\varphi_2 \in cl(\varphi)$.
- If $\bigcirc \varphi_1 \in cl(\varphi)$, then $\varphi_1 \in cl(\varphi)$.
- If $\varphi_1 \, U \, \varphi_2 \in cl(\varphi)$, then $\varphi_1 \in cl(\varphi)$ and $\varphi_2 \in cl(\varphi)$.
- If $\varphi_1 \, \tilde{U} \, \varphi_2 \in cl(\varphi)$, then $\varphi_1 \in cl(\varphi)$ and $\varphi_2 \in cl(\varphi)$.

Example:

$$\begin{aligned} cl(\Diamond \neg p) &= cl(\text{true}\ U \neg p) \\ &= \{\text{true}\ U \neg p,\ \neg p,\ \text{true}\} \\ &= \{\Diamond \neg p,\ \neg p,\ \text{true}\}\,. \end{aligned}$$

Note: Remember that in our definition of LTL, negation is only applied to atomic propositions (Negation Normal Form).

Sequence labeling rules

A labeling $\tau : \mathbb{N} \rightarrow 2^{cl(\varphi)}$ of a sequence $\pi : \mathbb{N} \rightarrow 2^P$ is valid if for every position $i \geq 0$ and formula φ' such that $\varphi' \in \tau(i)$, we have $\pi^i \models \varphi'$.

For this property to hold, it is sufficient to satisfy the following rules at every position $i \geq 0$:

Propositional rules:

- Rule 1: false $\notin \tau(i)$.
- Rule 2: For every $p \in P$:
  - If $p \in \tau(i)$, then $p \in \pi(i)$.
  - If $\neg p \in \tau(i)$, then $p \notin \pi(i)$.
- Rule 3: If $\varphi_1 \wedge \varphi_2 \in \tau(i)$, then $\varphi_1 \in \tau(i)$ and $\varphi_2 \in \tau(i)$.
- Rule 4: If $\varphi_1 \vee \varphi_2 \in \tau(i)$, then $\varphi_1 \in \tau(i)$ or $\varphi_2 \in \tau(i)$.

Notes:

- Rules 2–4 are "if" and not "if and only if" rules. In other words, a valid labeling is not necessarily maximal.

  For instance, it is allowed to have $\varphi_1 \in \tau(i)$ and $\varphi_2 \in \tau(i)$ without having $\varphi_1 \wedge \varphi_2 \in \tau(i)$.

- Rule 2 implies that $\tau(i)$ cannot contain both p and $\neg p$, for any atomic proposition $p \in P$.

Temporal rules:

The rule for the *Next* operator is easy:

- Rule 5: If $\bigcirc \varphi_1 \in \tau(i)$, then $\varphi_1 \in \tau(i+1)$.

For the *Until* and *Release* operators, the situation is more tricky, since their semantics involve the whole sequence. They can however be reduced to local conditions as follows:

- $\varphi_1 \, U \, \varphi_2 = (\varphi_2 \vee (\varphi_1 \wedge \bigcirc (\varphi_1 \, U \, \varphi_2)))$
- $\varphi_1 \, \tilde{U} \, \varphi_2 = (\varphi_2 \wedge (\varphi_1 \vee \bigcirc (\varphi_1 \, \tilde{U} \, \varphi_2)))$

We then obtain the following labeling rules:

- Rule 6: If $\varphi_1 \, U \, \varphi_2 \in \tau(i)$, then either
  - $\varphi_2 \in \tau(i)$, or
  - $\varphi_1 \in \tau(i)$ and $\varphi_1 \, U \, \varphi_2 \in \tau(i+1)$.
- Rule 7: If $\varphi_1 \, \tilde{U} \, \varphi_2 \in \tau(i)$, then
  - $\varphi_2 \in \tau(i)$, and
  - either $\varphi_1 \in \tau(i)$, or $\varphi_1 \, \tilde{U} \, \varphi_2 \in \tau(i+1)$.

Rule 7 is sufficient for establishing the validity of the labeling. Rule 6, on the other hand, is incomplete: it does not force the existence of a future position in which φ_2 holds. An additional rule is therefore needed:

- Rule 8: If $\varphi_1 \, U \, \varphi_2 \in \tau(i)$, then there exists $j \geq i$ such that $\varphi_2 \in \tau(j)$.

Correctness of the rules

Theorem: Let φ be a formula build from a set P of atomic propositions, and let $\pi : \mathbb{N} \to 2^P$. We have $\pi \models \varphi$ if and only if there exists a labeling $\tau : \mathbb{N} \to 2^{cl(\varphi)}$ of π that satisfies Rules 1–8, and that is such that $\varphi \in \tau(0)$.

Proof sketch:

- *If τ satisfies Rules 1–8, then for every $i \geq 0$ and $\varphi' \in \tau(i)$, we have $\pi^i \models \varphi'$.*

  This can be established by induction on the structure of the formulas in $cl(\varphi)$:

  - The base case directly follows from Rules 1–2.
  - The inductive cases are handled by reasoning about the semantics of the operators.

    Example: If φ' is of the form $\varphi_1 \, U \, \varphi_2$:

    * By Rule 8, there exists $j \geq i$ such that $\varphi_2 \in \tau(j)$. By inductive hypothesis, this implies $\pi^j \models \varphi_2$.
    * Consider the smallest such j. For $k = i, i+1, \ldots, j-1$, we have $\varphi_2 \notin \tau(k)$, hence by repeated applications of Rule 6, $\varphi_1 \in \tau(k)$ and $\varphi_1 \, U \, \varphi_2 \in \tau(k+1)$. We therefore have $\pi^k \models \varphi_1$ by inductive hypothesis, which leads to $\pi^i \models \varphi'$.

- *If $\pi \models \varphi$, then there exists a labeling τ that satisfies Rules 1–8 and such that $\varphi \in \tau(0)$.*

  Such a labeling can be obtained by defining for all $i \geq 0$

  $$\tau(i) = \{\varphi' \in cl(\varphi) \mid \pi^i \models \varphi'\}.$$

  Indeed:

  – Since $\pi \models \varphi$, we have $\varphi \in \tau(0)$.

  – The semantics of LTL ensure that Rules 1–8 are satisfied at all positions.

Automaton construction

Principle: An automaton accepting the sequences that satisfy a given LTL formula φ over a set P of atomic propositions can be obtained by

- associating its states with the subsets of $cl(\varphi)$ that satisfy Rules 1 and 3–4,
- defining its transitions so as to satisfy Rules 2 and 5–7,
- enforcing Rule 8 with the help of the accepting condition.

Formal construction: One builds a generalized Büchi automaton $(S, \Sigma, \delta, S_0, F)$ as follows.

- Its set of states S is the set of subsets s of $cl(\varphi)$ that satisfy
  - false $\notin s$
  - if $\varphi_1 \wedge \varphi_2 \in s$, then $\varphi_1 \in s$ and $\varphi_2 \in s$, and
  - if $\varphi_1 \vee \varphi_2 \in s$, then $\varphi_1 \in s$ or $\varphi_2 \in s$.
- Its alphabet Σ is 2^P.

- Its transition relation δ is such that $(s, a, s') \in \delta$ iff all the following conditions are satisfied:
  - For every $p \in P$:
    - if $p \in s$, then $p \in a$,
    - if $\neg p \in s$, then $p \notin a$.
  - If $\bigcirc \varphi_1 \in s$, then $\varphi_1 \in s'$.
  - If $\varphi_1 \, U \, \varphi_2 \in s$, then either
    - $\varphi_2 \in s$, or
    - $\varphi_1 \in s$ and $\varphi_1 \, U \, \varphi_2 \in s'$.
  - If $\varphi_1 \, \tilde{U} \, \varphi_2 \in s$, then
    - $\varphi_2 \in s$, and
    - either $\varphi_1 \in s$, or $\varphi_1 \, \tilde{U} \, \varphi_2 \in s'$.
- Its set of initial states is $S_0 = \{s \in S \mid \varphi \in s\}$.

- Its accepting set F is obtained as follows:
  - We call an eventuality formula every formula of the form $\varphi_1 \, U \, \varphi_2$ that belongs to $cl(\varphi)$. We have to ensure that every state that contains such a formula is eventually followed by a state that contains φ_2.
  - The transition relation is such that from every state in which $\varphi_1 \, U \, \varphi_2$ appears, this formula keeps appearing until φ_2 appears.
  - It is thus sufficient to consider as accepting the runs of the automaton that visit infinitely often either
    * a state in which $\varphi_1 \, U \, \varphi_2$ does not appear, or
    * a state in which $\varphi_1 \, U \, \varphi_2$ and φ_2 both appear.
  - Let $\varphi_{1,1} \, U \, \varphi_{2,1}, \varphi_{1,2} \, U \, \varphi_{2,2}, \ldots, \varphi_{1,k} \, U \, \varphi_{2,k}$ be all the eventuality formulas in $cl(\varphi)$. We have $F = \{F_1, F_2, \ldots, F_k\}$, where for each i such that $1 \leq i \leq k$:

$$F_i = \{s \in S \mid \{\varphi_{1,i} \, U \, \varphi_{2,i}, \varphi_{2,i}\} \subseteq s \ \vee \ \varphi_{1,i} \, U \, \varphi_{2,i} \notin s\}.$$

Improving the construction

The previous construction generates an automaton that is needlessly complicated. The following simplifications are aimed at reducing its size without affecting its accepted language:

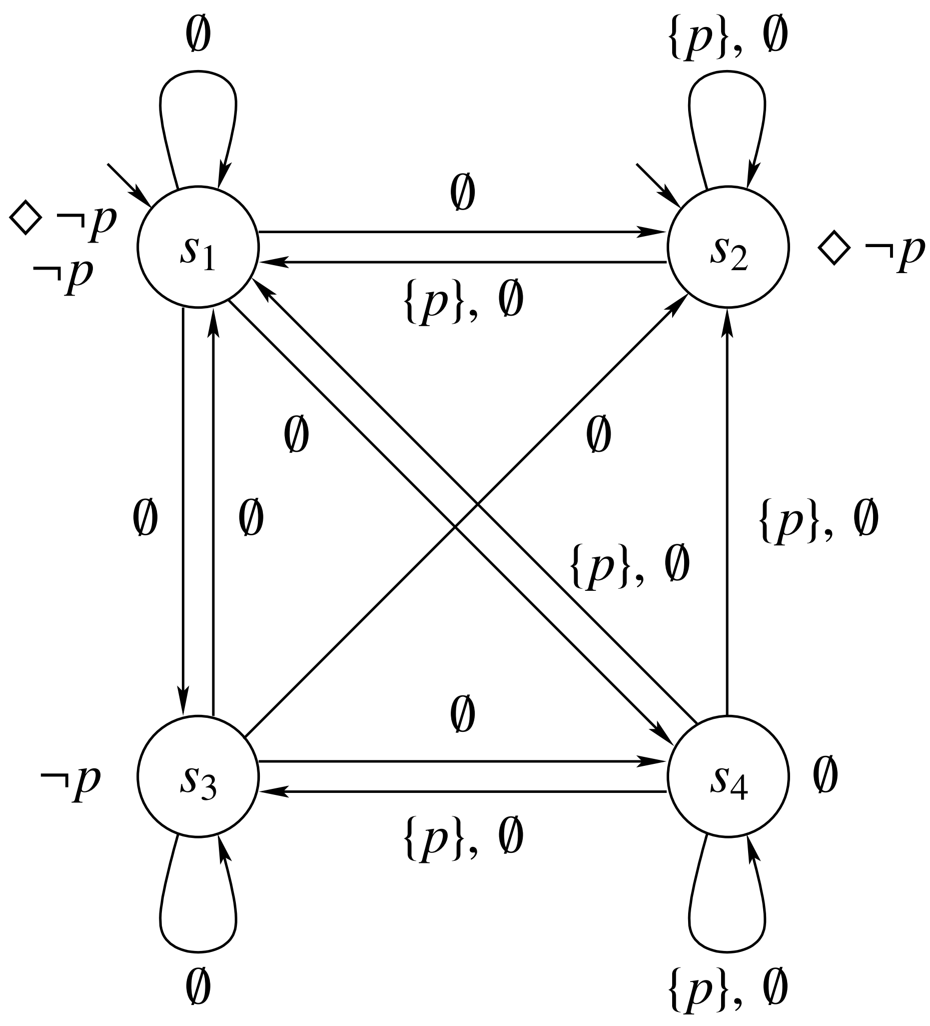
- We can consider implicitly that for every $s \in S$, we have true $\in s$. (Indeed, any sequence satisfies true.)

- For every $p \in P$, the states that contain both p and $\neg p$ do not need to be considered. (Indeed, such states cannot have outgoing transitions.) More generally, states with inconsistent formulas can be dropped.

- The unreachable states of the automaton can be discarded.

Example

$\varphi = \Diamond \neg p = \text{true}\, U \neg p \qquad cl(\varphi) = \{\Diamond \neg p,\ \neg p,\ \text{true}\}$



$F = \{\{s_1,\ s_3,\ s_4\}\}$

Removing redundant transitions

The automaton can be further simplified thanks to the following property:

> If two states s and s' are such that $s' \subset s$, and s' has at least one outgoing transition, then every infinite sequence that can be read from s can also be read from s'.

As a consequence, if there exist two transitions (s_1, a, s) and (s_1, a, s') for such s and s', the former is redundant and can be discarded. However, this is only correct if all sequences accepted from s can also be accepted from s'. We thus have the following rule:

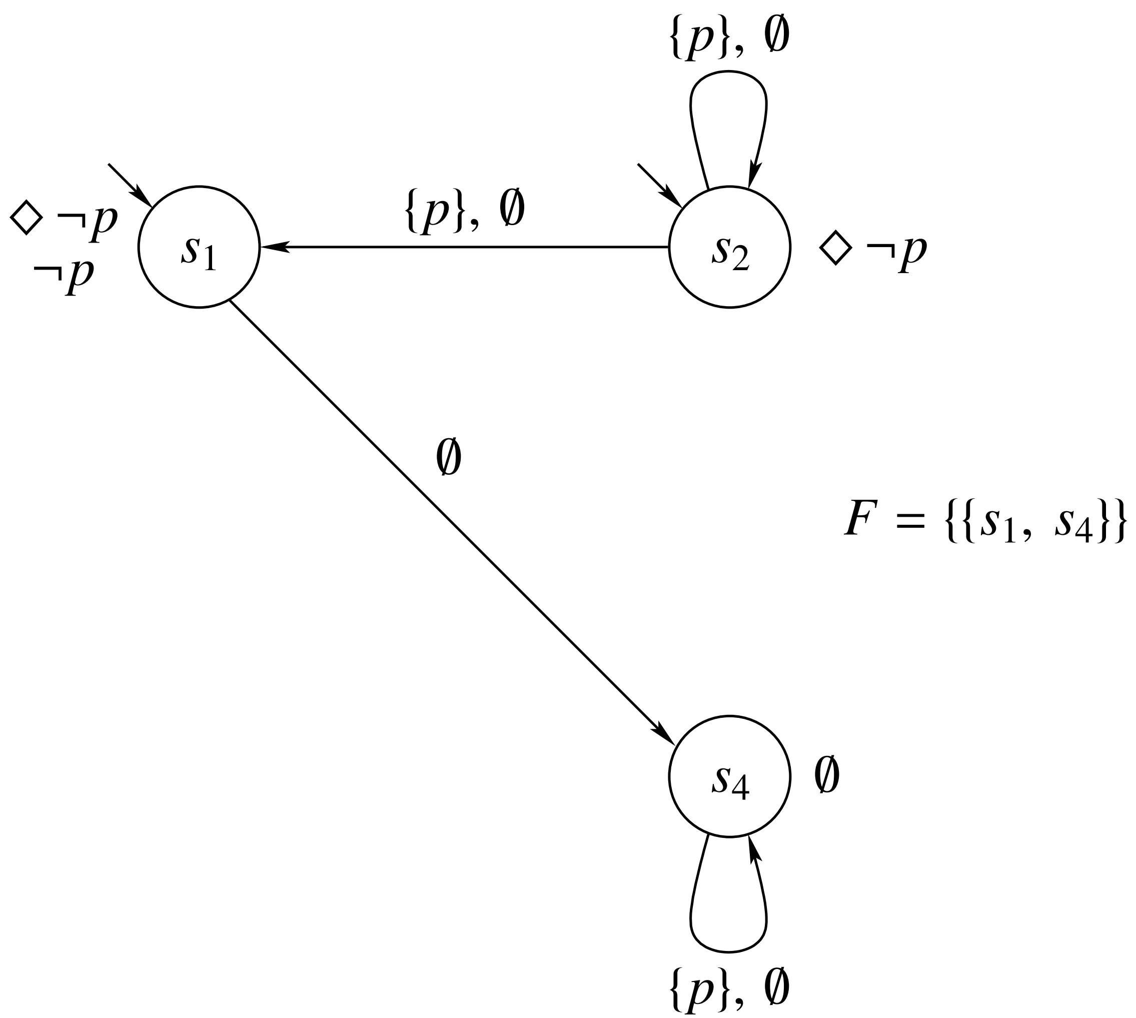
Simplification rule: If there exist two transitions (s_1, a, s) and (s_1, a, s') such that

- $s' \subset s$,
- s' has at least one outgoing transition, and
- for every eventuality formula $\varphi_1 \, U \, \varphi_2 \in s$, if $\varphi_1 \, U \, \varphi_2 \in s'$ and $\varphi_2 \in s$, then also $\varphi_2 \in s'$,

then the transition (s_1, a, s) can be omitted.

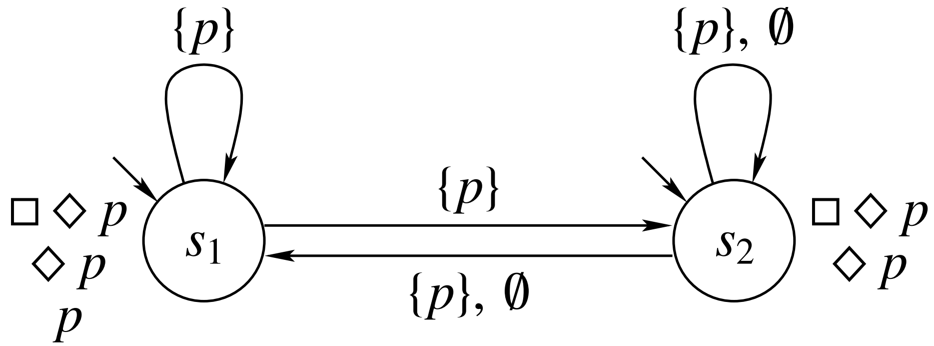
Illustration

$$\varphi = \Diamond \neg p$$



Example 2

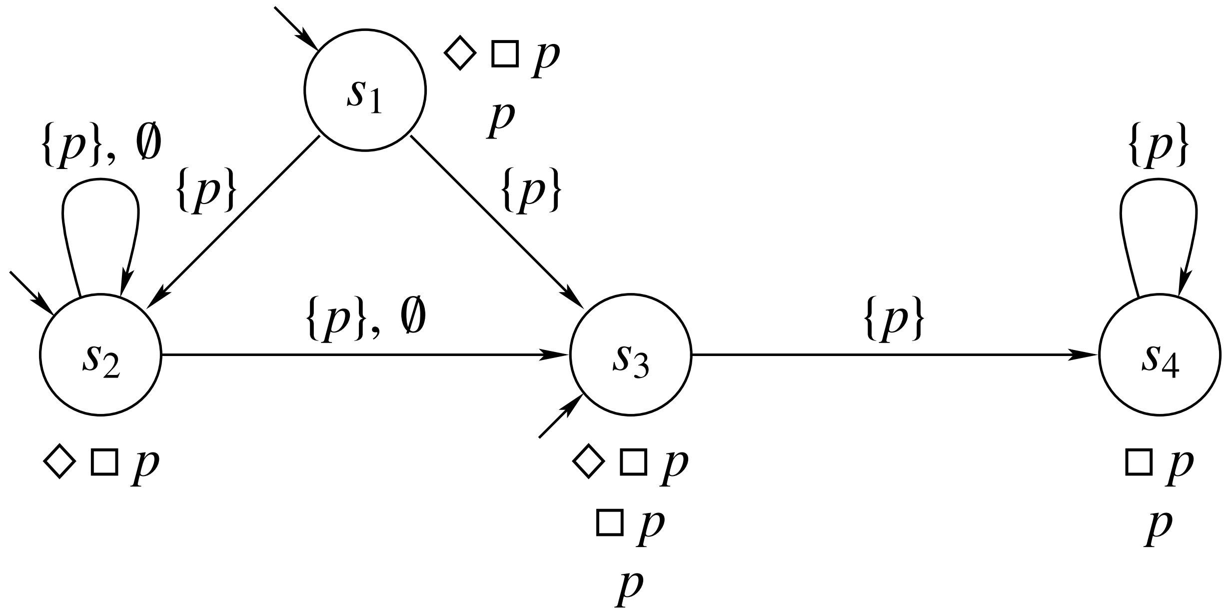
$\varphi = \Box \Diamond p = \text{false}\ \tilde{U}\ (\text{true}\ U\ p) \qquad cl(\varphi) = \{\Box \Diamond p,\ \Diamond p,\ p,\ \text{true}, \text{false}\}$



$F = \{\{s_1\}\}$

Example 3

$\varphi = \Diamond \Box p = \text{true}\ U\ (\text{false}\ \tilde{U}\ p) \qquad cl(\varphi) = \{\Diamond \Box p,\ \Box p,\ p,\ \text{true}, \text{false}\}$



$F = \{\{s_3,\ s_4\}\}$

Further improvements

The procedure for generating an infinite-word automaton from a LTL formula can still be improved in many ways:

- The automaton's states can be generated by need.
- States that are syntactically distinct but semantically identical can be merged.

  Example: $\{\varphi_1\, U\, \varphi_2, \varphi_2\} \equiv \{\varphi_2\}$.
- LTL formulas can be simplified before the construction.

  Example: $\bigcirc \Box \Diamond \varphi \equiv \Box \Diamond \varphi$.
- ...

References

- Most of the explanations in this chapter are borrowed from

P. Wolper, *Constructing Automata from Temporal Logic Formulas: A Tutorial*. Lectures on Formal Methods and Performance Analysis, LNCS 2090, pp. 261–277, 2001.

- The following articles develop efficient constructions:

R. Gerth, D. Peled, M. Y. Vardi, P. Wolper, *Simple on-the-fly automatic verification of linear temporal logic*. Proc. 15th Workshop on Protocol Specification, Testing, and Verification, pp. 3–18, 1995.

F. Somenzi, R. Bloem, *Efficient Büchi automata from LTL formulae*. Proc. 12th Intl. Conference on Computer-Aided Verification, LNCS 1633, pp. 247–263, 2000.

P. Gastin, D. Oddoux, *Fast LTL to Büchi automata translation*. Proc. 13th Intl. Conference on Computer-Aided Verification, LNCS 2102, pp. 53–65, 2001.

- There even exists a formally verified version of the construction algorithm:

  A. Schimpf, S. Merz, J.-G. Smaus, *Construction of Büchi automata for LTL model checking verified in Isabelle/HOL*. Proc. 22nd Intl. Conference on Theorem Proving in Higher-Order Logics, LNCS 5674, pp. 242–439, 2009.

Chapter 5

State-space exploration

The verification approach

- The central part of the verification process consists in computing the transition system corresponding to the FCS to be analyzed.

  Note: In practice, only the reachable states need to be computed.

- One can then check properties by examining this transition system.

  The properties that can be checked include the reachability of a given state or set of states, the absence of deadlocks, the correctness of an invariant, . . .

Depth-first search

Goal: Developing an algorithm that computes the transition system (S, Σ, s_0, T) corresponding to the reachable state-space of a formal concurrent system $\{\mathcal{P}, \mathcal{M}, \mathcal{T}\}$.

Mechanisms:

- The algorithm explores the states by means of a depth-first search, using a stack St and a data structure H for memorizing the set of visited states.

  Note: In actual implementations, the structure H usually takes the form of a hash table.

- A function $enabled(s)$ returns the transitions of $\mathcal{T}$ that are enabled in a state $s \in S$.

- A function $succ(s, t)$ returns, for each $s \in S$ and $t \in enabled(s)$, the state $s' \in S$ such that $(s, n(t), s') \in T$.

State-space exploration algorithm

```
procedure explore(s):
    St.push(s)
    H := H ∪ {s}
    for all t ∈ enabled(s):
        s′ := succ(s, t)
        if s′ ∉ H then:
            explore(s′)
    St.pop()

St := [ ]
H := ∅
explore(s0)
```

Notes:

- The reachable states of the generated transition system are those that belong to H. The transitions are the triples $(s, n(t), s')$ that correspond to the s, t and s' considered at Line 5 of the algorithm.

- The stack St does not affect the execution of the algorithm. If an erroneous state is reached, its purpose is to provide an execution trace that is helpful for understanding how such a state can be reached.

- Observer processes can be added for checking more elaborate properties, such as properties expressed in linear-time temporal logic.

- The transition relation does not need to be stored: verification is performed on-the-fly while exploring the state space.

Checking LTL properties

Principles:

- To check that a transition system satisfies a LTL property, one needs to check that all its behaviors satisfy this property (in other words, are models of this property).
- To obtain a model-checking procedure, one can add an observer process corresponding to the infinite-word automaton obtained from the property.

Problem: Such an automaton is non-deterministic, and accepts a behavior if there exists some computation for which it accepts. How can we check that for all behaviors of a transition system, the property is satisfied?

Solution: First complement the property to be checked!

Model-checking algorithm for LTL

Goal: Checking whether all executions of a given FCS satisfy a given LTL property φ.

Solution:

1. Build an infinite-word automaton $\mathcal{A}_{\neg\varphi}$ that accepts exactly all the infinite sequences that do not satisfy φ.

2. Add $\mathcal{A}_{\neg\varphi}$ as an observer process to the FCS. If needed, add also observer processes for the fairness conditions.

3. Explore the corresponding transition system with a depth-first search.

   Note: The generated transition system will have a generalized Büchi acceptance condition.

4. Check whether the resulting transition system is empty or not, i.e., whether it accepts at least one infinite sequence.

   An accepted infinite sequence corresponds to a behavior of the system that does not satisfy the property φ.

Note: We have not yet discussed how to check with a depth-first search the emptiness of the language accepted by a generalized Büchi automaton.

The goal is to be able to carry out this operation without explicitly computing the strongly connected components of the transition system.

Safety properties

Consider a property φ such that the corresponding automaton $\mathcal{A}_\varphi$ admits the generalized Büchi acceptance condition $F = \emptyset$.

This means that every infinite sequence that can be read by the automaton is accepted. In other words, the only way that a sequence cannot be accepted is by reaching a state from which no suitable transition can be followed.

Note: Since $\mathcal{A}_\varphi$ does not impose an acceptance condition, it can be determinized with the same algorithm as for finite-word automata.

The complement automaton $\mathcal{A}_{\neg\varphi}$ then has a single accepting state from which all sequences are accepted. In other words, once this state is reached, the sequence is accepted whatever happens next. The automaton $\mathcal{A}_{\neg\varphi}$ thus behaves like a finite-word one.

As a consequence:

- Fairness conditions are not necessary.
- The emptiness test reduces to a simple reachability test.

Remember that a LTL property can be seen as a set of sequences (or words): those that satisfy it.

Definition: A LTL property is a safety property if it corresponds to a set of sequences of the form

$$\Sigma^\omega \setminus L \cdot \Sigma^\omega,$$

where L is a (finite-word) regular language.

Liveness properties

We have seen that to falsify a safety property, it is enough to follow a finite prefix of a sequence.

Question: What are the properties that cannot be falsified by a finite prefix, i.e., such that every finite prefix can always be continued to form an accepted sequence?

Definition: A LTL property is a liveness property if it corresponds to a set L of sequences such that

$$\forall w \in \Sigma^* : \exists w' \in \Sigma^\omega : w\, w' \in L.$$

Note: One can prove that every property expressible by an infinite-word automaton is equivalent to the conjunction of a safety property and a liveness property.

Checking the nonemptiness of infinite-word automata

Principle: A Büchi automaton accepts a nonempty language if it has an accepting state that is reachable from an initial state, as well as reachable from itself by a path of non-zero length.

Note: Generalized Büchi automata can be handled by converting them to Büchi ones.

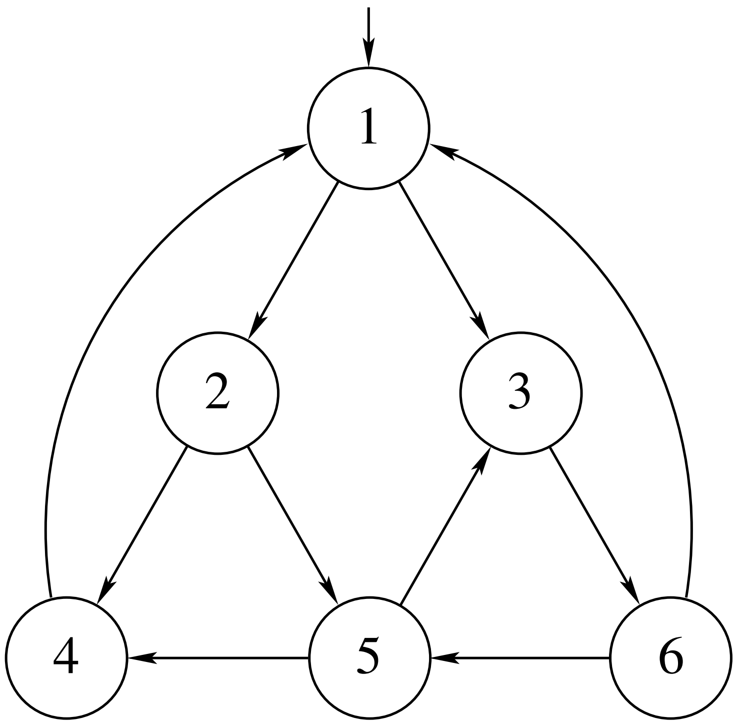
Algorithm:

1. Perform a depth-first search of the reachable states.
2. During this search, build a postorder list $Q = [s_1, s_2, \ldots, s_k]$ of the reachable states that are accepting, where s_1 is the first visited accepting state and s_k the last.
3. Perform a second search from the states in Q.

Notes:

- In a postorder traversal of a graph, the children of a node appear before this node.

  Example:



$$Q = [4, 6, 3, 5, 2, 1]$$

- For the correctness proof, we will also consider another postorder Q' containing all visited states (not only accepting ones as Q).

The second search

Goal: Check whether Q contains a state that is is reachable from itself by a path of non-zero length.

Algorithm:

1. Start the search from s_1.

2. The search from s_i is finished when either

   - s_i is reached, or

   - there is no more reachable state to visit.

   In the first case, the nonemptiness check has succeeded. In the second case, if $i < k$, we restart the search from s_{i+1} but without reconsidering the states that have already been visited during the searches from $s_1, s_2, \ldots, s_i$.

3. If all the s_i have been searched without success, then the nonemptiness check has failed.

Notes:

1. During this second search, each node of the automaton is visited at most once.

2. The first and second searches can be interleaved.

Why does it work?

Lemma: If $j < i$ and s_i is reachable from s_j, then s_j is reachable from itself.

Proof: Consider a path from s_j to s_i.

1. If no node on this path had been visited (by the second search) before s_j, then the search would have reached s_i from s_j, and s_i would have appeared before s_j in the postorder.

   Thus, the path necessarily contains a state that has been visited before s_j. Let s be the last such state in the path.

2. The state s cannot appear before s_j in the postorder Q', since s_i would then appear before s_j as well.

3. Since s has been visited before s_j, but appears after s_j in the postorder Q', it must be an ancestor of s_j.

4. Thus, s_j can reach one of its ancestors, and is thus reachable from itself.

Theorem: The second search correctly detects whether there exists some s_i that is reachable from itself.

Proof: The proof is by induction on the value of i:

1. For $i = 1$, the second search explores exhaustively all the states reachable from s_1.

2. For $i > 1$, if there exists a path from s_i to itself, then this path cannot contain a state reachable from some s_j with $j < i$.

   Indeed, in such a case, s_i would be reachable from s_j, and by the Lemma, s_j would be reachable from itself. By induction, this should have been detected in a prior step.

Complexity of LTL model-checking

- Linear (NLOGSPACE) in the size of the transition system.
- Exponential (PSPACE) in the size of the FCS.
- Exponential (PSPACE) in the size of the LTL formula.

Improvements to the state-space exploration procedure

- There exist advanced hash techniques for minimizing the amount of data that needs to be stored in the set H of all visited states.

  Some of these techniques are probabilistic, and may (infrequently) miss some reachable states.

- Partial-order methods have been developed for avoiding to explore redundant parts of the state-space, such as the sequences produced by the interleaving of independent transitions.

- There exist model checking algorithms suited for properties expressed in more elaborate temporal logics.

- ...

Notes: Implementations are available, e.g. the Spin tool (`https://spinroot.com`).

Chapter 6

Symbolic verification methods

Introduction

The state-space exploration algorithms studied in Chapter 5 have the advantage of being simple and general, but there are not able to handle very large state spaces.

One possibility of overcoming this limitation is to express verification as a logical problem:

- The transition system of the program under analysis defines a Kripke structure $\mathcal{K} = (P, W, R, W_0, L)$, where
  - P is a finite set of atomic propositions.
  - W is a finite set of states.
  - $R \subseteq W \times W$ is a transition relation between states, that is left-total, i.e., for each $w \in W$, there exists $w' \in W$ such that $(w, w') \in R$.

    Note: Deadlock situations can be modeled by transitions from a state to itself.
  - $W_0 \subseteq W$ is a set of initial states.
  - $L : W \rightarrow 2^P$ is a labeling of the states with sets of atomic propositions.

- The verification problem then reduces to checking properties of $\mathcal{K}$.
- The approach consists in performing such checks by logical methods, i.e., by representing and manipulating $\mathcal{K}$ and the properties of interest by formulas.

Principles:

- Since W is finite, each state $w \in W$ can be encoded with a fixed number n of Boolean values forming a bit vector $b_1 b_2 \dots b_n$.
- A set of states can then be represented by a Boolean function over $b_1, b_2, \dots, b_n$.

  In particular:

  - The set W_0 of initial states can be represented by a function I such that $I(b_1, b_2, \dots, b_n) =$ true iff $b_1 b_2 \dots b_n$ encodes a state w that belongs to W_0.
  - For each $p \in P$, the set of states w such that $p \in L(w)$ can be represented by a function p such that $p(b_1, b_2, \dots, b_n) =$ true iff $b_1 b_2 \dots b_n$ encodes a state w such that $p \in L(w)$.

- Similarly, the transition relation R can be represented by a Boolean function over $2n$ variables, such that $R(b_1, b_2, \ldots, b_n, b'_1, b'_2, \ldots, b'_n) =$ true iff $b_1 b_2 \ldots b_n$ encodes a state w and $b'_1 b'_2 \ldots b'_n$ encodes a state w' such that $R(w, w')$ holds.

- Properties of $\mathcal{K}$ can then be checked by computing with those Boolean functions.

Problems to address:

- How can we check properties (in particular, safety or LTL ones) of a Kripke structure represented by Boolean functions? What computations are required?

- What operations do these computations have to perform on Boolean functions?

- What is a good data structure for representing Boolean functions, that allows the required operations to be performed efficiently?

A description language for Kripke structures

The functions I, R, and p for each $p \in P$, that represent a Kripke structure $\mathcal{K} = (P, W, R, W_0, L)$, are expressed over Boolean variables.

In order to describe more simply the operations that need to be performed, we introduce a logical language based on states. This language is built from the following elements:

- A set $V = \{x, y, z, \ldots\}$ of variables over the domain W, i.e., the values taken by these variables are states.
- An interpreted predicate $I(x)$ that is true iff the value of x belongs to W_0, i.e., if it corresponds to an initial state.
- An interpreted predicate $p(x)$ for each $p \in P$, that is true iff the value of x is a state w such that $p \in L(w)$.
- An interpreted predicate $R(x, y)$ that is true iff the values of x and y are (respectively) states w and w' such that $R(w, w')$ holds.

- A set $V' = \{X, Y, Z, \ldots\}$ of uninterpreted predicate variables, each of those predicates having an arbitrary number of arguments.

  If needed for clarity, the fact that the predicate variable X admits k arguments (i.e., that X stands for a k-ary predicate) will be made explicit by writing X^k.

We are now ready to define a first version of our formalism:

- An interpreted predicate or a predicate variable applied to the required number of variables (corresponding to its number of arguments) is a formula.

- If φ_1 and φ_2 are formulas, then $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$ and $\neg\varphi_1$ are formulas as well.

- If φ is a formula and x is a variable, then $\exists x\, \varphi$ and $\forall x\, \varphi$ are formulas.

Examples:

- This formula expresses that every transition originating from a state in which p is true leads to a state in which q is false:

  $$\forall x\, \forall y\, (p(x) \wedge R(x, y) \ \Rightarrow\ \neg q(y))$$

  (Note: As usual, $\varphi_1 \ \Rightarrow\ \varphi_2$ is a shorthand for $\neg\varphi_1 \vee \varphi_2$.)

- This formula expresses that a state in which p is true is reachable in one step from the state corresponding to x:

$$\exists y\,(p(y) \land\ R(x, y))$$

- This formula expresses that the predicate corresponding to X is satisfied by all initial states as well as those reachable in one step from an initial state:

$$\forall x\,(X(x)\ \Leftrightarrow\ I(x) \lor \exists y\,(I(y) \land R(y, x)))$$

Note: $\varphi_1\ \Leftrightarrow\ \varphi_2$ is a shorthand for $(\varphi_1 \land \varphi_2) \lor (\neg\varphi_1 \land \neg\varphi_2)$.

Recursive definition of predicates

Our goal is now to extend our logical formalism by introducing a mechanism for defining predicates recursively.

Principes:

- We first introduce a notation for naming explicitly the free variables in a formula: if φ is a formula and x_1, x_2, ..., x_k are its free first-order variables, then

  $$\lambda x_1, x_2, \ldots x_k \, \varphi$$

  is a definition for a predicate Π with k arguments.

  If φ has no free second-order variable, then $\Pi(w_1, w_2, \ldots, w_k)$ is true, for some $w_1, w_2, \ldots, w_k \in W$, iff the formula φ in which the free variable x_1 takes the value w_1, the free variable x_2 takes the value w_2, and so on, is valid.

- A predicate definition $\lambda x_1, x_2, \ldots x_k \, \varphi$ is said to be monotone in a predicate variable X if X appears under the scope of an even number of negations in φ.

Intuitively, when a predicate definition for a predicate Π is monotone in X, modifying the predicate represented by X by adding new argument values for which this predicate is true leads to new argument values for which Π is true.

Formally, if the predicate B corresponding to X is replaced by B' such that

$$\forall x_1, x_2, \ldots\ (B(x_1, x_2, \ldots) \Rightarrow B'(x_1, x_2, \ldots)),$$

then the predicate Π expressed by the predicate definition becomes Π' such that

$$\forall x_1, x_2, \ldots\ (\Pi(x_1, x_2, \ldots) \Rightarrow \Pi'(x_1, x_2, \ldots)).$$

- If Λ is a k-ary predicate definition that is monotone in X^k, then $\mu X\, \Lambda$ is also a k-ary predicate definition, that corresponds to the least fixpoint of X with respect to Λ.

  This least fixpoint is defined as the smallest subset S of W^k such that if X takes the value S (meaning that $X(x_1, \ldots, x_k) =$ true iff $(x_1, \ldots, x_k) \in S$), then the predicate defined by Λ takes the value S as well.

Example: An unary predicate expressing that its argument is a reachable state can be expressed as follows:

$$\mu X\,\lambda x\,(I(x) \vee \exists y\,(X(y) \wedge R(y, x)))$$

Notes:

– Let $\Lambda :\ \lambda\, x_1, \ldots, x_k\, \varphi$ be a predicate definition that is monotone in X^k, and let $[\![\Lambda]\!]_B \subseteq W^k$ denote the predicate defined by Λ when X takes the value B, with $B \subseteq W^k$.

  The predicate defined by

$$\mu X\,\lambda\, x_1, \ldots, x_k\, \varphi$$

  is the limit of the sequence

$$B_0 \ \subseteq\ B_1 \ \subseteq\ B_2 \ \subseteq\ B_3 \ \subseteq\ \cdots\ ,$$

  where $B_0 = \emptyset$, and for each $i > 0$, $B_i = [\![\Lambda]\!]_{B_{i-1}}$.

– There also exists a notion of greatest fixpoint for a variable X that is monotone in a predicate definition Λ, denoted by $\nu X\,\Lambda$. (It will not be used in this chapter.)

- If Λ is a k-ary predicate definition, then $\Lambda(x_1, x_2, \ldots, x_k)$, where x_1, x_2, $\ldots$, x_k are variables, is a formula.

Example: The following formula expresses that there exists a reachable state in which p is true:

$$\exists z\,(p(z) \wedge\ (\mu X\,\lambda x\,(I(x) \vee \exists y\,(X(y) \wedge R(y, x))))(z))$$

Evaluating a formula of the description language

Problem: Let φ be a closed formula of the description language, i.e., without any free variable. How do we compute the truth value of φ?

Solution:

- If φ does not involve any fixpoint predicate definition, then since each $w \in W$ is encoded by n Boolean values, each variable $x \in V$ can be replaced by a set of n Boolean variables $b_1^x, b_2^x, \ldots, b_n^x$.

  Checking whether φ is valid then reduces to Boolean satisfiability:

  - Quantifications $\exists x\, \phi$ and $\forall x\, \phi$ can be replaced (respectively) by $\exists b_1^x\, \exists b_2^x \ldots \exists b_n^x\, \phi$ and $\forall b_1^x\, \forall b_2^x \ldots \forall b_n^x\, \phi$.
  - For a Boolean variable b, the quantified formulas $\exists b\, \phi(b)$ and $\forall b\, \phi(b)$ are respectively equivalent to

$$\phi(\text{true}) \lor \phi(\text{false})$$

  and

$$\phi(\text{true}) \land \phi(\text{false}).$$

- For computing the predicate corresponding to a least fixpoint definition

$$\mu X\, \lambda x_1, \ldots x_k\, \phi,$$

where ϕ is monotone in X, we explicitly develop the sequence

$$B_0 \subseteq B_1 \subseteq B_2 \subseteq B_3 \subseteq \cdots,$$

where $B_0 = \emptyset$, and for each $i > 0$, $B_i = [\![\lambda x_1, \ldots x_k\, \phi]\!]_{B_{i-1}}$, until it stabilizes.

Since W is finite, and $B_i \subseteq W$ for all i, this always happens after finitely many steps.

Computing reachable states symbolically

- As already discussed, a predicate corresponding to the set of reachable states of a symbolically defined Kripke structure is given by

$$\mu X\, \lambda x\, (I(x) \vee \exists y\, (X(y) \wedge R(y, x))).$$

  Notes:

  - Evaluating this predicate definition requires a number of steps that is bounded by the diameter of the Kripke structure.
  - In practice, the Boolean formulas appearing in the computation can become huge.

- To check if there exists a reachable state in which p is true, we can evaluate the formula

$$\exists z\, (I(z) \wedge (\mu X\, \lambda x\, (p(x) \vee \exists y\, (X(y) \wedge R(x, y))))(z)),$$

  which amounts to a backward search through the Kripke structure.

  This provides a method for checking safety properties.

Checking LTL properties

To check an LTL property, one adds an observer corresponding to the negation of this property to the system. One then obtains a Kripke structure with accepting states, for which we have to check repeated reachability.

Principes:

- The transitive closure R^+ of the reachability relation R can be computed thanks to the following predicate definition:

$$\mu R^+ \, \lambda x, y \, (R(x, y) \lor \exists z \, (R(x, z) \land R^+(z, y)))$$

- If $p \in P$ indicates accepting states, i.e., p is true in a state $w \in W$ iff w is accepting, then the predicate definition

$$\lambda x \, (p(x) \land R^+(x, x))$$

  characterizes the set of accepting states that are (non-trivially) reachable from themselves.

- Checking that there exists an accepting run then amounts to evaluating the formula

$$\exists z \, (p(z) \land R^+(z, z) \land (\mu X \, \lambda x \, (I(x) \lor \exists y \, (X(y) \land R(y, x))))(z)).$$

Speeding up the computation

The following predicate definition can be used to speed up the computation of R^+:

$$\mu R^+ \, \lambda x, y \, (R(x, y) \vee \exists z \, (R^+(x, z) \wedge R^+(z, y)))$$

Indeed, at each step of the fixpoint computation, this formula doubles the length of the paths that are considered.

However, in practice, the size of the generated formulas can become too large for the computation to succeed.

Required Boolean operations

To implement the method that we have described, we need a data structure for representing and manipulating Boolean formulas.

The following operations need to be computable on the representation of Boolean formulas:

1. Taking Boolean combinations and negations of formulas.
2. Applying existential and universal quantifiers.
3. Checking implication between two formulas (to terminate a fixpoint computation).
4. Computing the truth value of a closed formula (to get the result of the verification).

Operations (1) and (2) are immediate if the corresponding operators are in the language. However, performing (3) and (4) then becomes difficult.

A better strategy consists in representing formulas in a normal form from which (3) and (4) are easy, and to reduce formulas to this normal form each time (1) and (2) are performed.

Binary decision diagrams

A Binary Decision Diagram (BDD), or Ordered Binary Decision Diagram (OBDD), is a data structure for representing symbolically Boolean formulas with a fixed number of free variables (or, equivalently, Boolean functions with a fixed arity).

Main properties:

- BDDs depend on a total order that must be defined over the set of free variables.
- For a given order, each formula admits an easily computable normal form that is unique up to isomorphism.
- Algorithms are available for performing all the operations of interest over formulas represented by BDDs.
- BDDs can be much more concise that the formulas that they represent.

Principles of BDDs

Consider a Boolean function $f(b_1, b_2, \ldots, b_k)$ with k arguments b_1, b_2, . . . , b_k taken in that order.

This function can be represented by a full binary tree of depth k in which

- for all $i \in 1, \ldots, k$, the interior nodes at depth $i - 1$ are labeled by b_i.

  Note: The root has the depth 0, and the leaves the depth k.

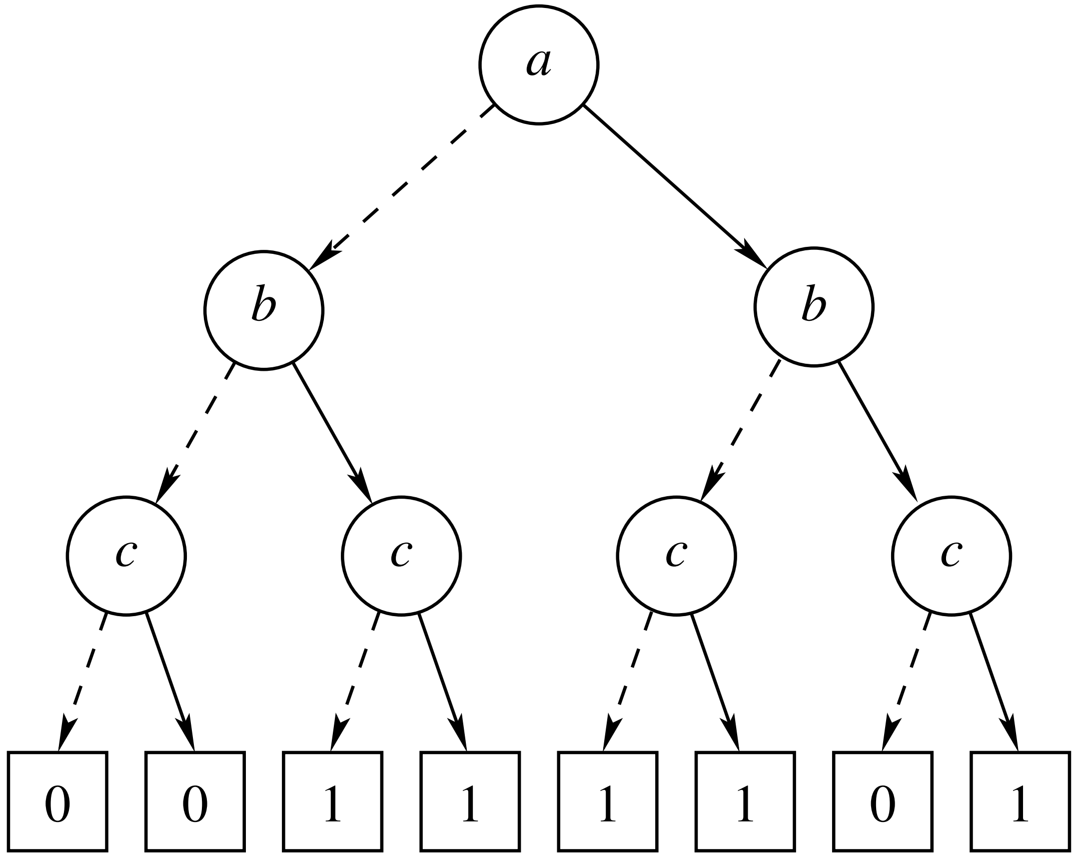
- each interior node has one outgoing transition labeled by 0 (false), and another one by 1 (true).

- the leaves are labeled by either 0 or 1. The label of a leaf provides the truth value of the function for input values that correspond to the label of paths leading to this leaf.

Example: For the function

$$f(a, b, c) = (a \vee b) \wedge (\neg a \vee \neg b \vee c),$$

we obtain the following tree.



Note: Edges labeled by 0 are denoted by dashed lines, and those labeled by 1 by solid lines.

Reducing BDDs

The following rules can be used to reduce BDDs:

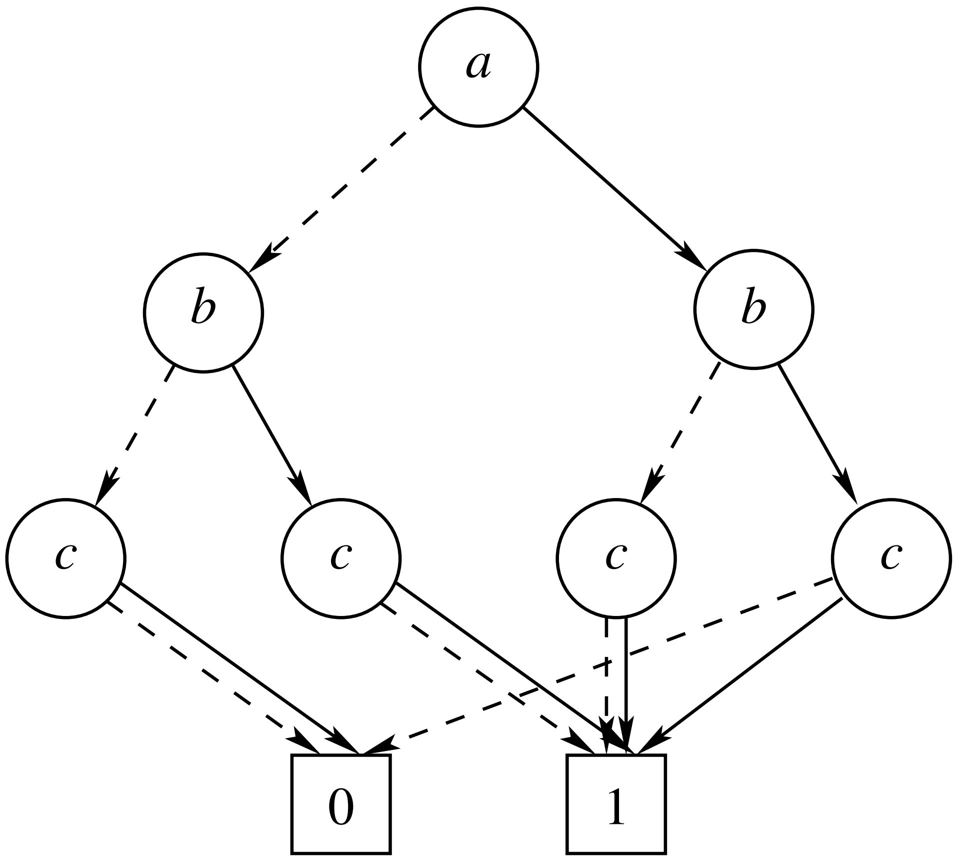
- *Eliminate redundant leaf nodes*: In other words, keep at most one leaf node labeled by 1 and at most one labeled by 0.
- *Remove duplicate interior nodes*: If two nodes labeled by the same variable have identical successors, eliminate one of these nodes, and redirect its incoming edges to the other.
- *Remove redundant tests*: If the two successors of a node are identical, eliminate that node and redirect its incoming edges to its unique successor.
- *Remove unreachable nodes*.

These rules are applied until no further reduction is possible.

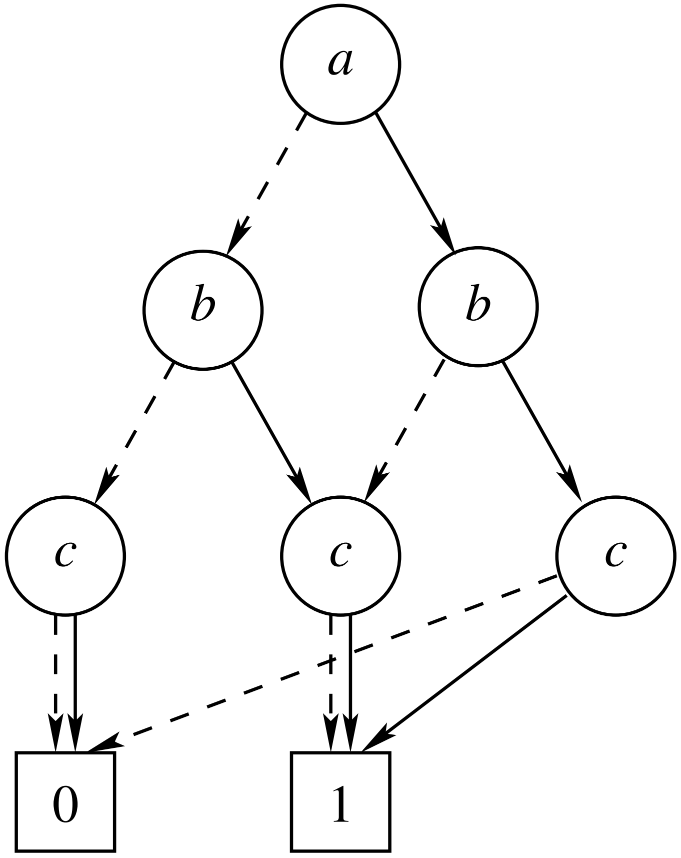
Theorem: For a given Boolean function and variable order, the maximally reduced BDD is unique up to isomorphism.

Illustration

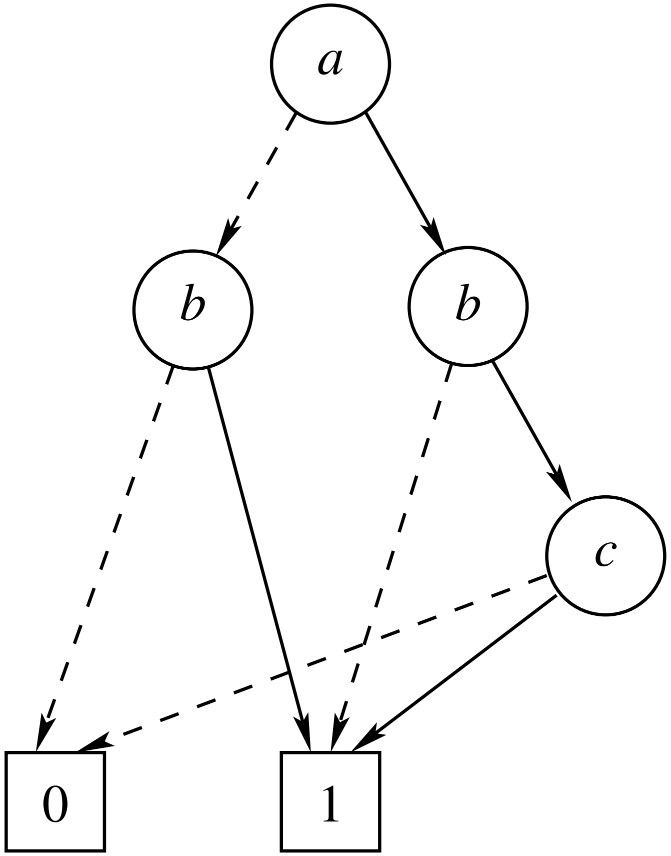
Applying the first rule to the BDD for $f(a,b,c) = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$ yields



After applying the second rule, we get



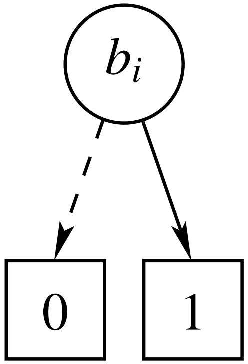
Finally, the third rule yields



which is the maximally reduced BDD representing f.

Elementary BDDs

- A BDD representing the function $f(b_1, b_2, \ldots, b_k) = b_i$ is easy to construct:



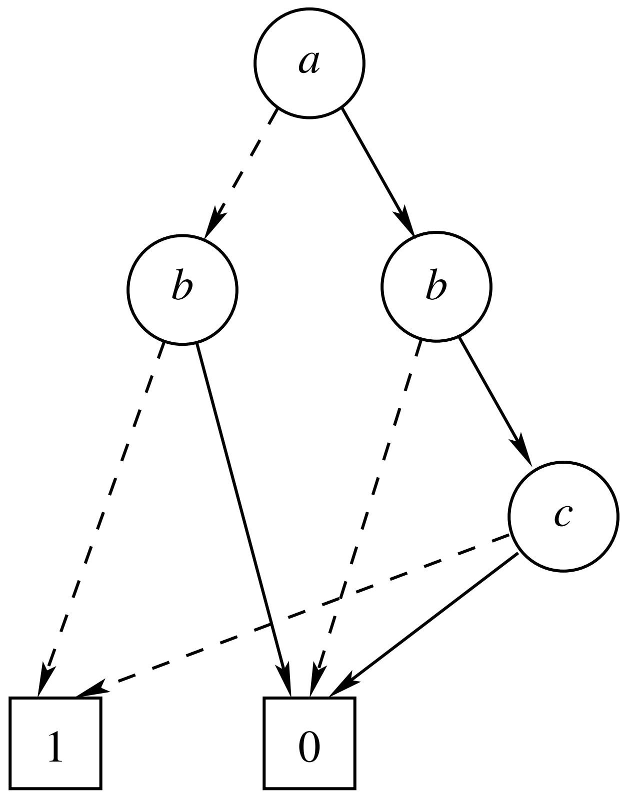
- The functions $f(b_1, b_2, \ldots, b_k) =$ true and $f(b_1, b_2, \ldots, b_k) =$ false are represented by BDDs that are only composed of one leaf node:

Operations on BDDs

Complementation:

Complementing a Boolean function represented by a BDD simply amounts to complementing the label of the leaves.

Example: $f(a, b, c) = \neg((a \vee b) \wedge (\neg a \vee \neg b \vee c))$



Binary Boolean operations ($\wedge$ and $\vee$):

These operations are performed by constructing a BDD that simulates the combined operation of the two BDDs representing the arguments, on the same input.

Formally, we represent a BDD by a tuple $(V, N, n_0, L_0, L_1, var, lo, hi)$, where

- V is a finite totally ordered set of variables.
- N is a finite set of nodes.
- $n_0 \in N$ is the root node.
- $L_0 \subseteq N$ and $L_1 \subseteq N$ are sets of respectively 0- and 1-leaves, such that $L_0 \cap L_1 = \emptyset$.
- $var : N \to V$ is a labeling of the nodes by variables. This labeling is such that for every leaf node $n \in L_0 \cup L_1$, its label $var(n)$ is a dummy maximal variable.
- $lo : N \to N$ and $hi : N \to N$ are respectively the 0- and 1-successor functions. These functions are total, and such that
  - For every $n \in N \setminus (L_0 \cup L_1)$: $var(lo(n)) > var(n)$ and $var(hi(n)) > var(n)$.
  - For every $n \in L_0 \cup L_1$: $lo(n) = hi(n) = n$.

Given a BDD $\mathcal{B}_1 = (V, N_1, n_{0,1}, L_{0,1}, L_{1,1}, var_1, lo_1, hi_1)$ representing a Boolean function f_1 and a BDD $\mathcal{B}_2 = (V, N_2, n_{0,2}, L_{0,2}, L_{1,2}, var_2, lo_2, hi_2)$ representing a Boolean function f_2 over the same variables, a BDD $\mathcal{B} = (V, N, n_0, L_0, L_1, var, lo, hi)$ representing $f_1 \wedge f_2$ or $f_1 \vee f_2$ can be constructed as follows:

- $N = N_1 \times N_2$.
- $n_0 = (n_{0,1}, n_{0,2})$.
- L_0 and L_1 are defined according to the Boolean function to be computed:
  - For $f_1 \wedge f_2$: $L_0 = (L_{0,1} \cup L_{1,1}) \times L_{0,2} \cup L_{0,1} \times (L_{0,2} \cup L_{1,2})$, and $L_1 = L_{1,1} \times L_{1,2}$.
  - For $f_1 \vee f_2$: $L_0 = L_{0,1} \times L_{0,2}$ and $L_1 = (L_{0,1} \cup L_{1,1}) \times L_{1,2} \cup L_{1,1} \times (L_{0,2} \cup L_{1,2})$.
- For each $(n_1, n_2) \in N$, $var((n_1, n_2)) = \min(var_1(n_1),\ var_2(n_2))$.
- For each $(n_1, n_2) \in N$,

$$lo((n_1, n_2)) = \begin{cases} (lo(n_1), lo(n_2)) & \text{if } var_1(n_1) = var_2(n_2) \\ ((lo(n_1), n_2) & \text{if } var_1(n_1) < var_2(n_2) \\ ((n_1, lo(n_2)) & \text{if } var_1(n_1) > var_2(n_2). \end{cases}$$

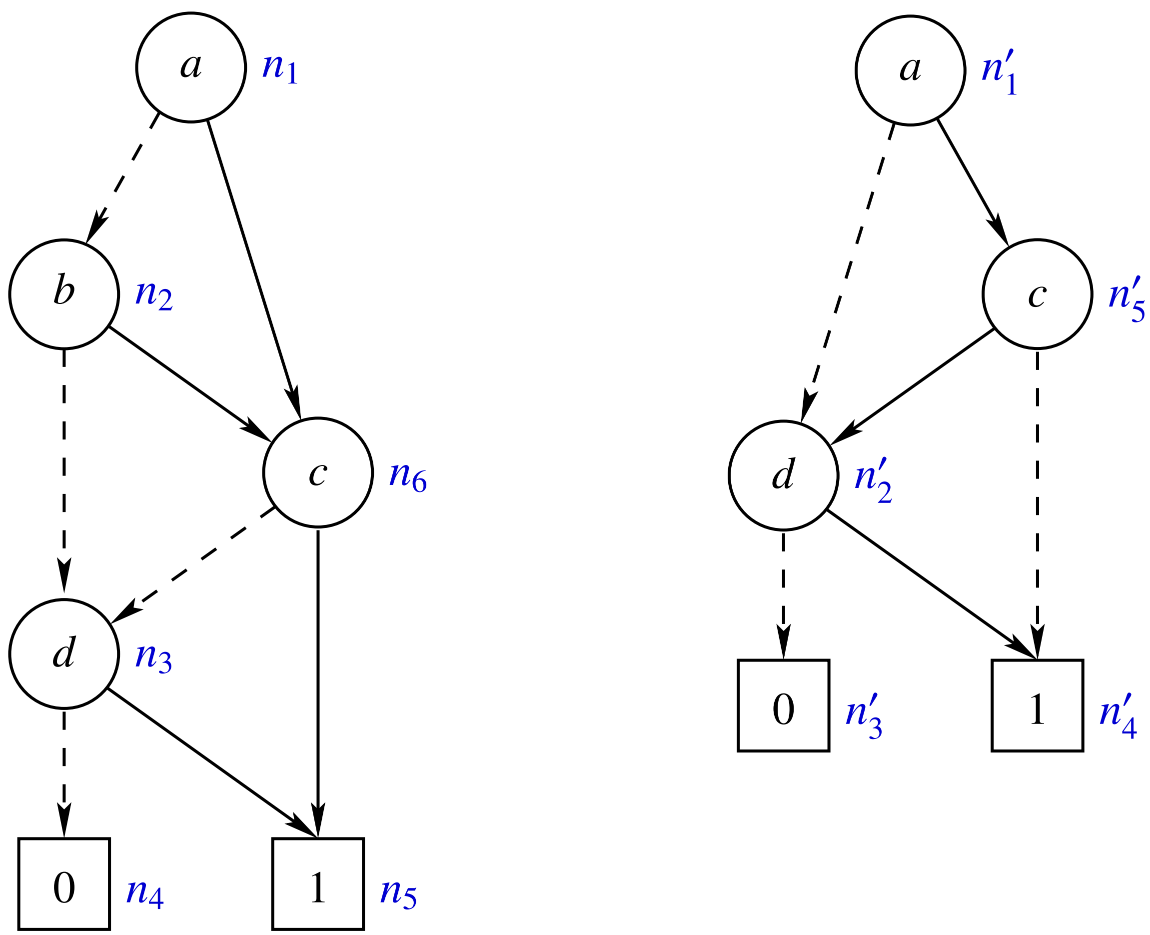
- For each $(n_1, n_2) \in N$,

$$hi((n_1, n_2)) = \begin{cases} (hi(n_1), hi(n_2)) & \text{if } var_1(n_1) = var_2(n_2) \\ ((hi(n_1), n_2) & \text{if } var_1(n_1) < var_2(n_2) \\ ((n_1, hi(n_2)) & \text{if } var_1(n_1) > var_2(n_2). \end{cases}$$

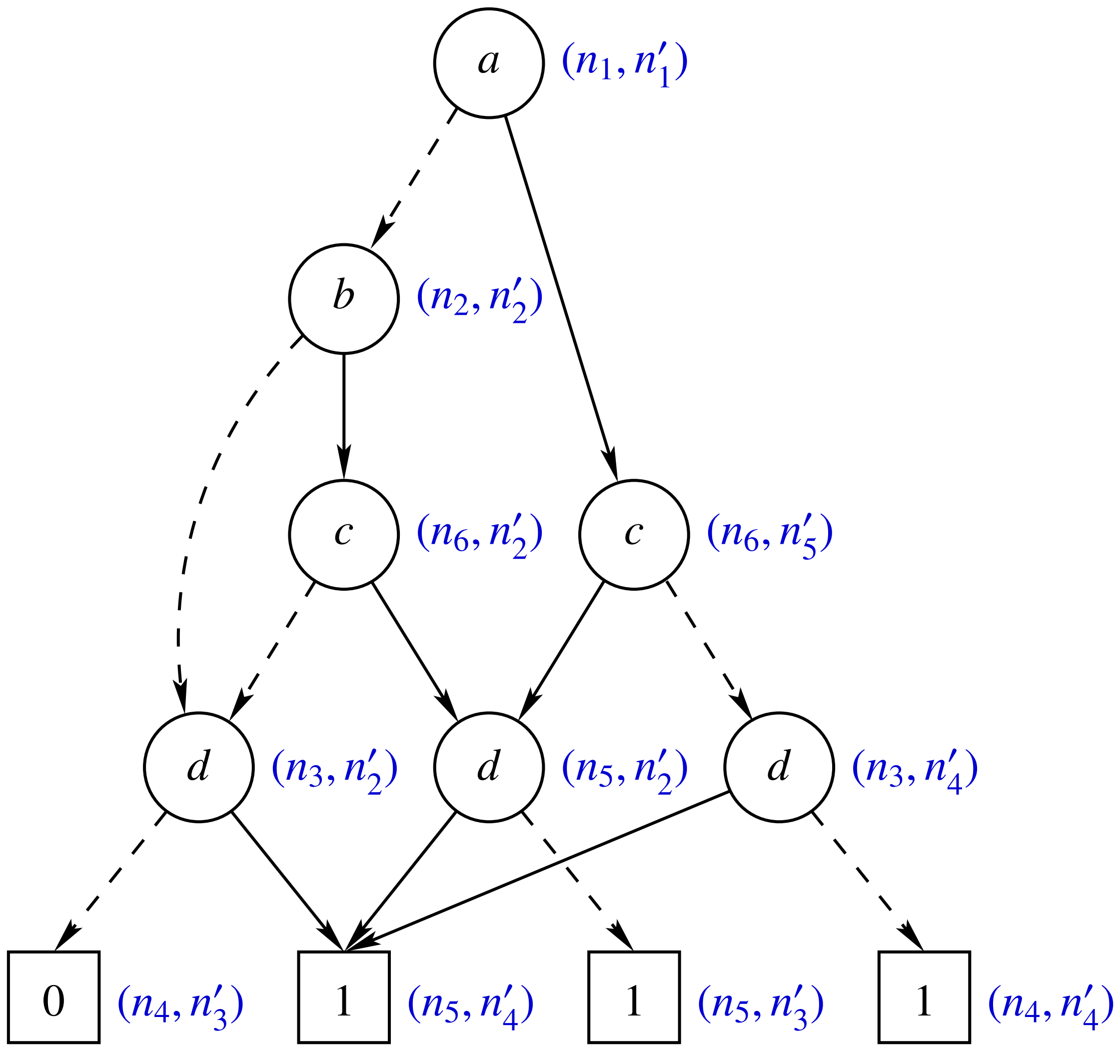
In practice, the BDD is constructed incrementally from its root node, and only reachable nodes are considered. The resulting BDD is then reduced to its normal form.

Example

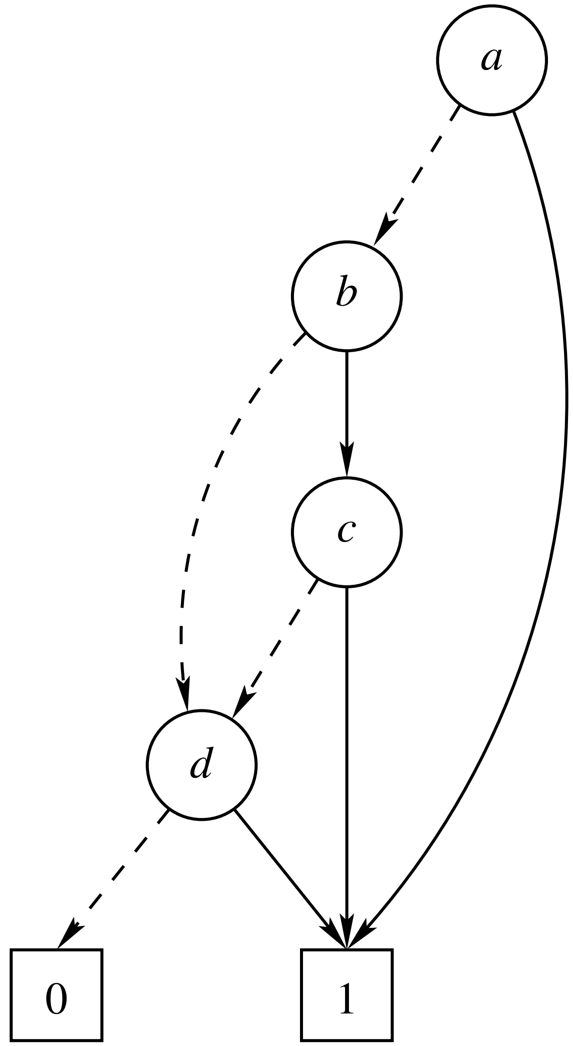
Computation of the disjunction of the Boolean functions represented by the two following BDDs.



Result of the construction:



After reduction:



Quantification:

To implement quantification, one uses the restrict operator that forces the value of a given variable to be 0 or 1.

Notation: For a function f and a variable x, the restrictions of f to the values 0 and 1 of x are (respectively) denoted by $f|_{x\leftarrow 0}$ and $f|_{x\leftarrow 1}$.

Computing $f|_{x\leftarrow 0}$ or $f|_{x\leftarrow 1}$ from a BDD representing f amounts to eliminating all nodes labeled by x, and redirecting their incoming edges to the appropriate successor.

For computing the effect of a quantification, one then uses the following identities:

- $\exists x\, f \;\equiv\; f|_{x\leftarrow 0} \vee f|_{x\leftarrow 1}$.
- $\forall x\, f \;\equiv\; f|_{x\leftarrow 0} \wedge f|_{x\leftarrow 1}$.

Implication:

To determine whether $f_1 \Rightarrow f_2$ holds, one computes the BDD for $\neg f_1 \vee f_2$, and then check that the resulting Boolean function is valid.

Checking for validity, satisfiability and unsatisfiability:

- Validity: All reachable leaf nodes are labeled by 1 (or equivalently, the reduced BDD is only composed of the leaf node 1).
- Satisfiability: At least one reachable leaf node is labeled by 1.
- Unsatisfiability: All reachable leaf nodes are labeled by 0 (or equivalently, the reduced BDD is only composed of the leaf node 0).

Complexity

- The reduction of a BDD $\mathcal{B}$ to its normal form can be performed in $O(|\mathcal{B}|)$ time.
- Complementation is $O(1)$.
- Binary Boolean operations applied to two BDDs $\mathcal{B}_1$ and $\mathcal{B}_2$ and testing their inclusion need $O(|\mathcal{B}_1|.|\mathcal{B}_2|)$ time.
- Applying a quantifier to a BDD $\mathcal{B}$ needs $O(|\mathcal{B}|^2)$ time.
- Tests for validity, satisfiability, and unsatisfiability run in $O(1)$ or $O(|\mathcal{B}|)$ time, depending on whether the BDD is or is not in normal form.

In practice:

- The variable ordering can have a huge impact on the size of BDDs.
- The state encoding is derived from the text of the program under analysis, hence the number of bits used is often much larger than the logarithm of the number of reachable states.