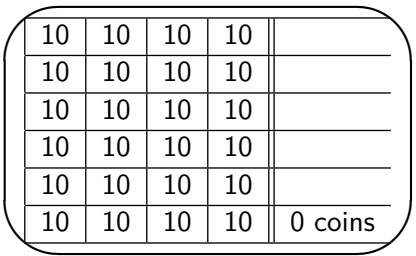
Exercise: Snack Dispenser

Snack dispenser has 6 levels with 4 slots each. Each slot can hold 10 items. Each item costs 2 CHF. The only way to operate a machine is to insert 1 CHF coin into temporary storage (one step), which must be done two times, then select the refreshment (in one step), which immediately dispenses the snack or cancels and returns coins if none is available in the slot, or if the stable coin storage is full. The machine can hold up to 500 CHF in its stable coin storage. It starts full with items but with no coins. Describe this transition system and estimate the cardinality of the set of all of its states. Can the stable storage ever hold 99 coins? 100 coins? 490 coins?



Exercise: Snack Dispenser

Snack dispenser has 6 levels with 4 slots each. Each slot can hold 10 items. Each item costs 2 CHF. The only way to operate a machine is to insert 1 CHF coin into temporary storage (one step), which must be done two times, then select the refreshment (in one step), which immediately dispenses the snack or cancels and returns coins if none is available in the slot, or if the stable coin storage is full. The machine can hold up to 500 CHF in its stable coin storage. It starts full with items but with no coins. Describe this transition system and estimate the cardinality of the set of all of its states. Can the stable storage ever hold 99 coins? 100 coins? 490 coins?

10	10	10	10	
10	10	10	10	
10	10	10	10	
10	10	10	10	
10	10	10	10	
10	10	10	10	0 coins

states: $10^{4 \cdot 6} \cdot 501 \cdot 2 = 10^{24} \cdot 1002 > 10^{27}$

Exercise: Snack Dispenser

Snack dispenser has 6 levels with 4 slots each. Each slot can hold 10 items. Each item costs 2 CHF. The only way to operate a machine is to insert 1 CHF coin into temporary storage (one step), which must be done two times, then select the refreshment (in one step), which immediately dispenses the snack or cancels and returns coins if none is available in the slot, or if the stable coin storage is full. The machine can hold up to 500 CHF in its stable coin storage. It starts full with items but with no coins. Describe this transition system and estimate the cardinality of the set of all of its states. Can the stable storage ever hold 99 coins? 100 coins? 490 coins?

10	10	10	10	
10	10	10	10	
10	10	10	10	
10	10	10	10	
10	10	10	10	
10	10	10	10	0 coins

states: $10^{4 \cdot 6} \cdot 501 \cdot 2 = 10^{24} \cdot 1002 > 10^{27}$

coins: $0 \to 2 \to \ldots \to 98 \to 100 \to 240 \cdot 2 = 480$

Sets of States (Reachable, Invariants) are Too Large to Store Explicitly

Approach: use formulas and data structure to represent them compactly
Use general formulas as in Stainless: expressive, but not very automated

For finite state systems: can get much more automation.

Two important algorithms:

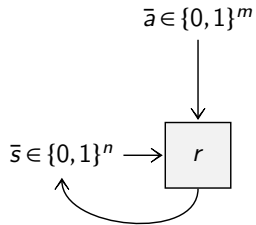
- bounded model checking using SAT solvers
- (reachability) model checking using binary decision diagrams (BDDs)

These algorithms play important role in model checking hardware designs and are basis for tools and more complex algorithms.

Encoding Finite Transition Systems with Bits: Sequential Circuit

Consider a deterministic finite-state transition system: $M = (S, I, r, A)$
If we pick $n \geq \log_2 |S|$ and $m \geq \log_2 |A|$, we can represent the finite-state transition system using boolean functions:

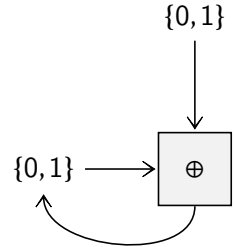
- each element of S as $\bar{s} \in \{0,1\}^n$, so $S = \{0,1\}^n$
- each element of A as $\bar{a} \in \{0,1\}^m$, so $A = \{0,1\}^m$
- initial states $I \subseteq S$ by the characteristic function $\{0,1\}^n \rightarrow \{0,1\}$
- deterministic transition relation $r \subseteq S \times A \times S$ as function $(S \times A) \rightarrow S$, that is, $\{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$



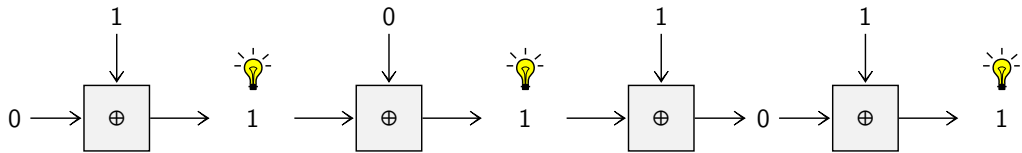
(For non-deterministic systems, we represent r as $(S \times A \times S) \rightarrow \{0,1\}$)

Example: Blinking Lights

- $S = \{0,1\}$ (1 = "light on")
- $A = \{0,1\}$ (1 = "toggle light")
- $I(s) = (s = 0)$
- $r(s,a) = s \oplus a$

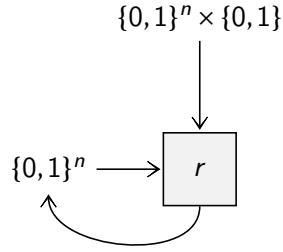


Example trace:

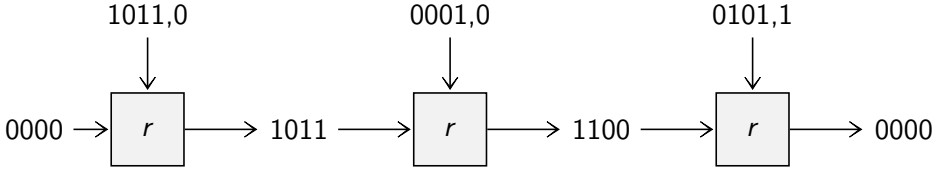


Example: Accumulator with Add and Clear Commands

- $S = \{0,1\}^n$ (value of accumulator)
- $A = \{0,1\}^n \times \{0,1\}$ (number to add, clear signal)
- $I(s) = (s = 0^n)$
- $r(s,(i,c)) =$ if (c) then 0 else $s +_n i$
  ($+_n$ is addition modulo 2^n)



Example trace:

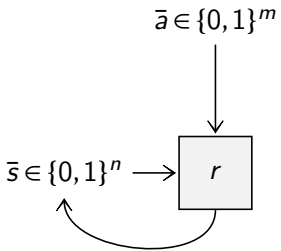


Encoding Finite Transition Systems with Bits: Sequential Circuit

Consider a deterministic finite-state transition system: $M = (S, I, r, A)$

If we pick $n \geq \log_2 |S|$ and $m \geq \log_2 |A|$, we can represent the finite-state transition system using boolean functions:

- each element of S as $\bar{s} \in \{0,1\}^n$, so $S = \{0,1\}^n$
- each element of A as $\bar{a} \in \{0,1\}^m$, so $A = \{0,1\}^m$
- initial states $I \subseteq S$ by the characteristic function $\{0,1\}^n \to \{0,1\}$
- deterministic transition relation $r \subseteq S \times A \times S$ as function $(S \times A \times S) \to S$, that is, $\{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$



How to represent boolean functions, like r, efficiently?

Boolean Function Representation: Table

Let $r : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$

List each of the $\{0,1\}^{n+m}$ inputs and specify the result

When $s, a \in \{0,1\}^4$ and represent addition of 4-bit non-negative integers modulo 16:

s	a	$r(s,a)$
0000	0000	0000
0000	0001	0001
0000	0010	0010
	⋯	
0001	0000	0001
0001	0001	0010
	⋯	
1111	1111	1110

$2^{4+4} = 256$ columns

In general, we cannot do better than truth table (there are that many different functions), but truth table always has bad representation size, even for functions like $+_4$.

Boolean Function Representation: Formulas

Let $r \subseteq \{0,1\}^n \times \{0,1\}^m \times \{0,1\}^n$
We represent the condition

$$((s_1,\ldots,s_n),(a_1,\ldots,a_m),(s'_1,\ldots,s'_m)) \in r$$

by writing a propositional formula with variables $s_1,\ldots,s_n$, $a_1,\ldots,a_m$, $s'_1,\ldots,s'_m$ that is true precisely when the tuple belongs to r.
If p is a propositional variable and $v \in \{0,1\}$ then we define p^v by $p^1 = p$ and $p^0 = \neg p$.
We can always represent r by a propositional formula in disjunctive normal form:

$$\bigvee_{((v_1,\ldots,v_n),(u_1,\ldots,u_m),(v'_1,\ldots,v'_n)) \in r} \left(\bigwedge_{1 \le i \le n} s_i^{v_i} \wedge \bigwedge_{1 \le i \le m} a_i^{u_i} \wedge \bigwedge_{1 \le i \le n} (s'_i)^{v'_i} \right)$$

so we do not lose on generality. Moreover, for many boolean functions, we can write down smaller formulas.

Propositional (Boolean) Logic

Propositional logic is a language for representing Boolean functions $f : \{0,1\}^n \to \{0,1\}$.

- sometimes we write $\bot$ for 0 and $\top$ for 1

Grammar of formulas:

$$P ::= x \mid 0 \mid 1 \mid P \wedge P \mid \neg P \mid P \vee P \mid P \oplus P \mid P \to P \mid P \longleftrightarrow P$$

where x denotes variables (identifiers). Corresponding Scala trees:

```
sealed abstract class Expr
case class Var(id: Identifier) extends Expr
case class BooleanLiteral(b: Boolean) extends Expr
case class And(e1: Expr, e2: Expr) extends Expr
case class Or(e1: Expr, e2: Expr) extends Expr
case class Not(e: Expr) extends Expr
...
```

Environment and Truth of a Formula

An environment e is a partial map from propositional variables to $\{0,1\}$
For vector of n boolean variables $\bar{p} = (p_1, \ldots, p_n)$ and $\bar{v} = (v_1, \ldots, v_n) \in \{0,1\}^n$, we denote $[\bar{p} \mapsto \bar{v}]$ the environment e given by $e(p_i) = v_i$ for $1 \leq i \leq n$.
We write $e \models F$, and define $[\![F]\!]_e = 1$, to denote that F is true in environment e, otherwise define $[\![F]\!]_e = 0$
Let $e = \{(a,1),(b,1),(c,0)\}$ and F be $a \wedge (\neg b \vee c)$. Then:

$$[\![a \wedge (\neg b \vee c)]\!]_e \;=\; e(a) \wedge (\neg e(b) \vee e(c)) \;=\; 1 \wedge (\neg 1 \vee 0) \;=\; 0$$

The general definition is recursive:

$$\begin{aligned} [\![x]\!]_e &= e(x) \\ [\![0]\!]_e &= 0 \\ [\![1]\!]_e &= 1 \\ [\![F_1 \wedge F_2]\!]_e &= [\![F_1]\!]_e \wedge [\![F_2]\!]_e \\ [\![\neg F_1]\!]_e &= \neg [\![F_1]\!]_e \end{aligned}$$

Note: $\wedge$ and $\neg$ on left and right are different things

Truth of a Formula in Scala

We can define it as interpret method on propositional expressions class:

```
def interpret(env: Map[Identifier, Boolean]): Boolean = this match {
  case Var(id) => env(id)
  case BooleanLiteral(b) => b
  case Equal(e1, e2) => e1.interpret(env) == e2.interpret(env)
  case Implies(e1, e2) => !e1.interpret(env) || e2.interpret(env)
  case And(e1, e2) => e1.interpret(env) && e2.interpret(env)
  case Or(e1, e2) => e1.interpret(env) || e2.interpret(env)
  case Xor(e1, e2) => e1.interpret(env) ^ e2.interpret(env)
  case Not(e) => !e.interpret(env)
}
```

Satisfiability Problem

Formula F is *satisfiable*, iff there **exists** e such that $[\![F]\!]_e = 1$.
Otherwise we call F *unsatisfiable*: when there does not exist e such that $[\![F]\!]_e = 1$, that is, for all e, $[\![F]\!]_e = 0$.
Example: let F be $a \wedge (\neg b \vee c)$. Then F is satisfiable, with e.g. $e = \{(a,1),(b,0),(c,0)\}$
Its negation of $\neg F$, is also satisfiable, with e.g. $e = \{(a,0),(b,0),(c,0)\}$

SAT is a problem: given a propositional formula, determine whether it is satisfiable.

The problem is decidable because given F we can compute its variables $FV(F)$ and it suffices to look at the 2^n environments for $n = FV(F)$. The problem is NP-complete, but useful heristics exist.

A SAT solver is a program that, given boolean formula F, either:

- returns **sat**, and, optionally, returns one environment e such that $[\![F]\!]_e = 1$, or
- returns **unsat** and, optionally, returns a **proof** that no satisfying assignment exists

Observation about Eliminating Variables

Let F, G be propositional formulas and c a propositional variable
Let $F[c := G]$ denote the result of replacing in F *each* occurrence of c by G:

$$\begin{array}{rcl} c[c := G] & = & G \\ (F_1 \wedge F_2)[c := G] & = & F_1[c := G] \wedge F_2[c := G] \\ (F_1 \vee F_2)[c := G] & = & F_1[c := G] \vee F_2[c := G] \\ (\neg F_1)[c := G] & = & \neg(F_1[c := G]) \end{array}$$

We also generalize to simultaneous replacement of many variables, $F[\bar{c} := \bar{G}]$
Then following formulas are equivalent (have same truth for all free variables):

- $F[c := G]$
- $\exists c.((c = G) \wedge F)$
- $\forall c.((c = G) \rightarrow F)$

Note: free variables are the variables occurring in the formula minus quantified ones (c)

Free Variables for Quantified Boolean Formulas

Quantified boolan formulas (QBF) are build from propositional variables and constants $0, 1$ using $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \exists, \forall$
(We also write $=$ for $\leftrightarrow$.) A boolean formula is a QBF without quantifiers $\forall, \exists$.
Definition of free variables of a formula:

$$\begin{aligned}
FV(v) &= \{v\} \text{ when } v \text{ is a propositional variable} \\
FV(F_1 \wedge F_2) &= FV(F_1) \cup FV(F_2) \\
FV(F_1 \vee F_2) &= FV(F_1) \cup FV(F_2) \\
FV(F_1 \rightarrow F_2) &= FV(F_1) \cup FV(F_2) \\
FV(\neg F_1) &= FV(F_1) \\
FV(\exists v.F_1) &= FV(F_1) \setminus \{v\} \\
FV(\forall v.F_1) &= FV(F_1) \setminus \{v\}
\end{aligned}$$

An environment e maps propositional variables to $\{0, 1\}$ (sometimes written $\{\bot, \top\}$)
For vector of n boolean variables $\bar{p} = (p_1, \ldots, p_n)$ and $\bar{v} = (v_1, \ldots, v_n) \in \{0, 1\}^n$, we denote $[\bar{p} \mapsto \bar{v}]$ the environment e given by $e(p_i) = v_i$ for $1 \leq i \leq n$.
We write $e \models F$ to denote that F is true in environment e.

Validity and Equivalence

Definition: Formula F is valid, iff for all e, $e \models F$.

Observation: F is valid iff $\neg F$ is unsatisfiable.

Definition: Formulas F and G are equivalent iff for every e that defines all variables in $FV(F) \cup FV(G)$, we have: $e \models F$ iff $e \models G$.

Observation: F and G are equivalent iff $F \leftrightarrow G$ is valid.

$\exists p.F$ is equivalent to $P[p := 0] \vee P[p := 1]$ whereas $\forall p.F$ to $P[p := 0] \wedge P[p := 1]$

Boolean Function Representation: Circuits

Formulas correspond to *trees*: variables are leaves, operations internal nodes.
More efficient representation that exploits sharing: directed acyclic graphs (DAGs).
We can view DAGs as formulas with *auxiliary variable* definitions.
Example for simple (ripple-carry) n-bit adder:

- input numbers: $s_1 \dots s_n$ and $a_1 \dots a_n$
- output: $s'_1 \dots s'_n$

The formula with auxiliary variables $c_1, \dots, c_{n+1}$:

$$c_1 = 0 \wedge \bigwedge_{i=1}^{n} \left(s'_i = s_i \oplus a_i \oplus c_i\right) \wedge \left(c_{i+1} = \left(s_i \wedge a_i\right) \vee \left(s_i \wedge c_i\right) \vee \left(a_i \wedge c_i\right)\right)$$

We can implement such definitions in hardware: route an output of one gate to multiple other gates.
To get back a tree: substitute all auxiliary variables c_i, but we get much bigger formula. Or, existentially quantify all auxiliary variables.

Formula Representation of Sequential Circuits

We represent sequential circuit as $C = (\bar{s}, \mathit{Init}, R, \bar{x}, \bar{a})$ where:

- $\bar{s} = (s_1, \ldots, s_n)$ is the vector of state variables
- Init is a boolean formula with $FV(\mathit{Init}) \subseteq \{s_1, \ldots, s_n\}$
- $\bar{a} = (a_1, \ldots, s_m)$ is the vector of input variables
- $\bar{x} = (x_1, \ldots, x_k)$ is the vector of auxiliary variables (for R)
- R is a boolean formula called transition formula, for which

$$FV(R) \subseteq \{s_1, \ldots, s_n, a_1, \ldots, a_m, x_1, \ldots, x_k, s_1', \ldots, s_n'\}$$

Transition system for C is (S, I, r, A) where $S = \{0,1\}^n$, $A = \{0,1\}^m$,

- $I = \{\bar{v} \in \{0,1\}^n \mid [\bar{s} \mapsto \bar{v}] \models \mathit{Init}\}$
- $r = \{(\bar{v}, \bar{u}, \bar{v}') \in \{0,1\}^{n+m+n} \mid [(\bar{s}, \bar{a}, \bar{s}') \mapsto (\bar{v}, \bar{u}, \bar{v}')] \models \exists \bar{x}.R\}$

Auxiliary variables $\bar{x}$ are treated as existentially quantified, can use conjucts $x_i = E(\bar{s}, \bar{a}, \bar{x})$ to express intermediate values.

Checking Inductive Invariant using SAT Queries

Given sequential circuit representation $C = (\bar{s}, Init, R, \bar{x}, \bar{a})$ and a formala Inv with $FV(Inv) \subseteq \{s_1, \ldots, s_n\}$, how do we check that Inv is an inductive invariant?

Checking Inductive Invariant using SAT Queries

Given sequential circuit representation $C = (\bar{s}, \mathit{Init}, R, \bar{x}, \bar{a})$ and a formala Inv with $FV(\mathit{Inv}) \subseteq \{s_1, \ldots, s_n\}$, how do we check that Inv is an inductive invariant?
Let us write *negations* of conditions "$\mathit{Init} \subseteq \mathit{Inv}$" and "$\mathit{Inv} \bullet r \subseteq \mathit{Inv}$"

- An initial state is not included in invariant:

$$\mathit{Init} \wedge \neg \mathit{Inv}$$

Checking Inductive Invariant using SAT Queries

Given sequential circuit representation $C = (\bar{s}, Init, R, \bar{x}, \bar{a})$ and a formala Inv with $FV(Inv) \subseteq \{s_1, \ldots, s_n\}$, how do we check that Inv is an inductive invariant?
Let us write *negations* of conditions "$Init \subseteq Inv$" and "$Inv \bullet r \subseteq Inv$"

- An initial state is not included in invariant:

$$Init \wedge \neg Inv$$

- There is a state satisfying invariant, leading to a state that breaks invariant:

$$\underbrace{Inv}_{\bar{s}} \wedge \underbrace{R}_{\bar{s},\bar{a},\bar{x},\bar{s}'} \wedge \underbrace{\neg Inv[\bar{s} := \bar{s}']}_{\bar{s}'}$$

  Note that $\bar{a}, \bar{x}$ variables are also existentially quantified, as they should be.

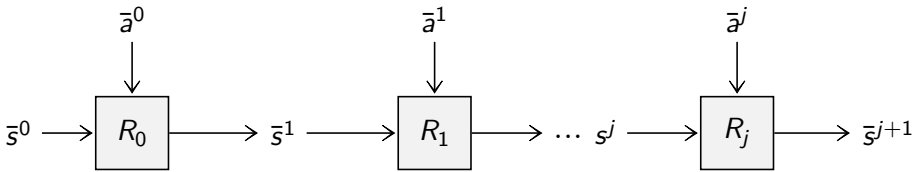
We can check if a formula is an inductive invariant using two queries to a SAT solver and making sure that they both return **unsat**.

Bounded Model Checking for Reachability

We construct a propositional formula T_j such that formula is satisfiable if and only if there exist a trace of length j starting from initial state that satisfies error formula E where $FV(E) \subseteq \{s_1, \ldots, s_n\}$.

$\bar{s}^i$ denotes state variables in step i.

$\bar{a}^i$ denotes inputs in step i.



$$T_j \;\equiv\; \mathit{Init}[\bar{s} := \bar{s}^0] \;\wedge\; \left(\bigwedge_{i=0}^{j-1} R_i\right) \;\wedge\; E[\bar{s} := \bar{s}^j]$$

where R_i is our transition formula, with variables renamed:

$$R_i \;\equiv\; R[(\bar{s}, \bar{a}, \bar{x}, \bar{s}') := (\bar{s}^i, \bar{a}^i, \bar{x}^i, \bar{s}^{i+1})]$$