CONVERTING BOOLEAN FUNCTIONS TO FINITE STATE MACHINES

DOMINIC VAN DER ZYPEN

1. Problem setting

We want to detect some events based on simple Yes/No rules that are connected via `AND` and `OR` gates.

The goal is to try to classify events into problematic vs non-problematic even if only partial information is available.

Ficticious example:

Suppose we know that an event is only problematic if

- the message transmitted contains the substring `''execute''`, **and**
- the port used is 8500.

Suppose we haven't analysed the message transmission yet, but we know that the message came through port 4112. Then by the above AND-rule, we can already classify as non-problematic and don't need to wait for the result of the analysis of the message.

2. Disjuncitve Normal Form DNF

Note that $\wedge$ denotes Boolean `AND` (mnemonic: $\wedge$ looks like 'A' as in AND), and $\vee$ denotes Boolean `OR` (mnemonic: $\vee$ looks like 'V' in Latin VEL meaning OR/AND which exactly captures the meaning of logical OR).

Any Boolean expression can be written in the form

$$(X_{1,1} \wedge \ldots \wedge X_{1,n_1}) \vee (Y_{2,1} \wedge \ldots \wedge Y_{2,n_2}) \vee \ldots \vee (X_{k,1} \wedge \ldots \wedge X_{k,n_k})$$

where every $X_{i,j}$ is either a basic variable or a negation of a basic variable.

We call this canonical form the *disjunctive normal form (DNF)*.

3. Simplifying assumptions

(1) We can get by using NO NEGATIONS as we formulate triggers positively.
(2) We assume our filter formula is given in *disjunctive normal form* (DNF) as introduced above. Any Boolean formula can be transformed into DNF, and it is easier to do so if the formula doesn't include negatives (see item 1).
(3) Per item to be analyzed, every trigger gets only set once to 0 or 1, there are no corrections or repetitions.

4. Finite state machine: definition

Formally, a *finite state machine (FSM)* is a 5-tuple $(\Sigma, S, s_0, \delta, F)$ where

- $\Sigma \neq \emptyset$ is the *input alphabet*,
- S is the *set of states*,
- $s_0 \in S$ is the *initial state*,
- $\delta : S \times \Sigma \to S$ is the *transition function*,
- $F \subseteq S$ is the set of *final states*. For "ever-running" machines, we may have $F = \emptyset$.

Recall that for any Boolean expression, we can convert it to a canonical form, the disjunctive normal form (DNF). So there is a big OR gate, and we have small AND gates, all at the same level, leading into the big OR gate. This is expressed with the formula we already saw:

$$(X_{1,1} \wedge \ldots \wedge X_{1,n_1}) \vee (Y_{2,1} \wedge \ldots \wedge Y_{2,n_2}) \vee \ldots \vee (X_{k,1} \wedge \ldots \wedge X_{k,n_k})$$

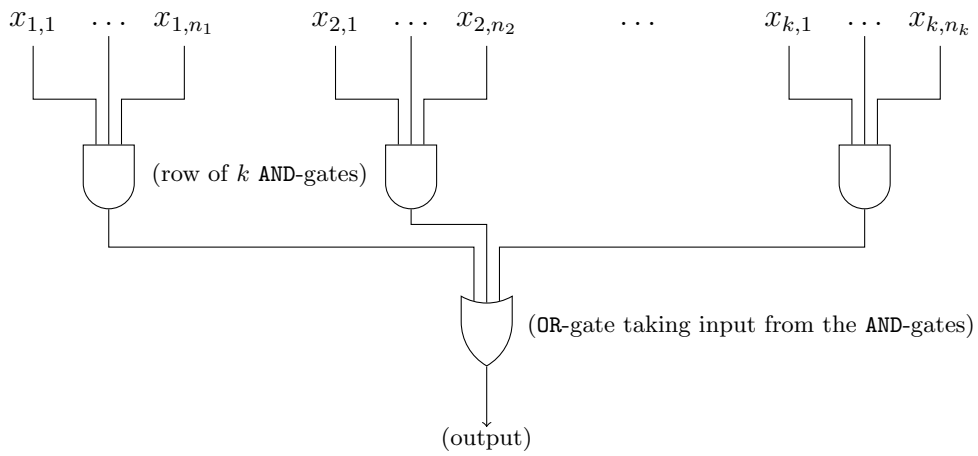
The goal for the remainder of this note is to build a finite state machine (FSM) that can evaluate a Boolean formula given in DNF even with only partial information - if that information is sufficient to determine the truth value of the whole formula.

5. Building a DNF FSM

There are several ways to build a finite state machine that adheres to the disjunctive normal form. We present one variant that appears quite natural - whether it will fit the business purposes, remains to be seen.

First we want to depict the DNF (disjunctive normal form) situation graphically. By $x_{i,j}$ we denote the inputs (triggers), some of which may

be set to TRUE, others to FALSE, and about some others the status may not be known.



Lemma 5.1. *It is easy to classify the output given the following kinds of partially available inputs:*

(1) *If for every* AND*-gate, we have at least one false input, the output is FAIL (false).*
(2) *If for at least one* AND*-gate, say* $\texttt{AND}_i$*, where* $i \in \{1, \ldots, k\}$*, all the inputs are TRUE, or formally, if*

$$x_{i,j} = 1 \text{ for all } j \in \{1, \ldots, n_i\}$$

then the output is HIT (true).
(3) *Otherwise, the output is undefined.*

Based on this easy classification, we will now give a mathematical description adhering to the definition of a finite state machine.

As depicted above, we have 1 big OR-gate, with k AND-gates are feeding into it. The AND-gate number i has n_i inputs which we call "triggers". Each trigger can be set to TRUE (T), FALSE (F), or it is undefined.

5.1. **Input alphabet Σ.** We set $\Sigma = \{1, \ldots, k\} \times \{T, F\}$. That is, an "input event" consists of the number of the AND-gate into which the corresponding trigger feeds, and the truth value indecating whether the trigger is set to T or F. So, for example input $(3, T)$ means that one trigger leading into AND-gate number 3 is set to true (T). (Note that any trigger can be set only **once**, see the section on simplifying assumptions above.)

5.2. **Set of states S.** First, for any positive integer $m \in \mathbb{N}$, we define

$$[n]^* := \{0, 1, \ldots, n\} \cup \{-1\}$$

and set

$$S = [n_1]^* \times [n_2]^* \times \cdots \times [n_k]^*.$$

So every state is a vector. We now give the interpretation of these state vectors: If $v \in S$, then if $v_i > 0$ means that v_i inputs have **not yet** been set to TRUE. If $v_i = 1$, this means that all but 1 inputs are set to TRUE, and if we receive another TRUE input for `AND`-gate i, then `AND`-gate i has all inputs set to TRUE and is TRUE itself, which also renders the output of the `OR`-gate TRUE. If we receive a FALSE input into `AND`-gate i, that is the member of the input alphabet is (i, F), then immediately we know that the output of `AND`-gate i must be FALSE. To indicate this, we set $v_i = -1$. Note that if our state vector is -1 at every position, that is $v_i = -1$ for all i, then the output of the `OR`-gate is FALSE too, and we have reached a terminal state.

Let's consider two examples:

- $k = 2$, and $v = (3, 1)$. This means we have 2 `AND`-gates and in the first gate, all except 3 entry points are TRUE, and for the second `AND`-gate, all except 1 entries have been set to TRUE.
- $k = 3$ and $v = (2, 1, -1)$. Similar as above, except `AND`-gate number 3 has seen one entry point being set to FALSE (which makes the whole `AND`-gate 3 output FALSE).

5.3. **Set of terminal states F.** $F \subseteq S$ is defined by

$$F := F_{\text{true}} \cup F_{\text{false}},$$

where

$$F_{\text{true}} = \big\{v \in S : \exists i \in \{1, \ldots, k\}(v_i = 0)\big\},$$

and

$$F_{\text{false}} = \big\{v \in S : \forall i \in \{1, \ldots k\}(v_i = -1)\big\} = \big\{(-1, -1, \ldots, -1)\big\}.$$

Note that F_{false} only has 1 element, the constant -1 vector, which is reached when every `AND`-gate has seen some FALSE input. A final state in F_{true} is reached when for some `AND`-gate, all the inputs have been set to true. Every member of $S \setminus F$ is considered to be an "undefined" or non-final state.

5.4. **Initial state $s_0 \in S$.** We set

$$s_0 = (n_1, n_2, \ldots, n_k).$$

Recall that this means that for every `AND`-gate, NO inputs have been set to TRUE, because we are at the beginning of the process.

5.5. **Transition function $\delta : S \times \Sigma \to S$.** And now we need to define the transition function $\delta : S \times \Sigma \to S$, which is the "clockwork" of the whole finite state machine.

- End state behaviour: Whenever $v \in F$ and $\sigma \in \Sigma$ we set $\delta(v, \sigma) = v$.
- Suppose $\sigma = (i, T) \in \Sigma$ where $i \in \{1, \ldots, k\}$, and $v \in S$.
  (1) If $v_i = -1$ we set $\delta(v, \sigma) = v$.
  (2) If $v_i \geq 1$ we set $\delta(v, \sigma) = v'$ where $v'_i = v_i - 1$ and $v'_j = v_j$ for all $j \in \{1, \ldots, k\} \setminus \{i\}$.
  (3) Note that we do not have to cover the case $v_i = 0$ as this is an end state, and we covered end states in the bullet point above.
- Suppose $\sigma = (i, F) \in \Sigma$ where $i \in \{1, \ldots, k\}$, and $v \in S$.
  (1) If $v_i = -1$ we set $\delta(v, \sigma) = v$.
  (2) If $v_i \geq 1$ we set $\delta(v, \sigma) = v'$ where $v'_i = -1$ and $v'_j = v_j$ for all $j \in \{1, \ldots, k\} \setminus \{i\}$.
  (3) Note that we do not have to cover the case $v_i = 0$ as this is an end state, and we covered end states in the bullet point above.

Size of state space: It is easy to see that for any $n \in \mathbb{N}$ we have

$$\operatorname{card}([n]^*) = \left|[n]^*\right| = n + 2.$$

So

$$\operatorname{card}(S) = \operatorname{card}\big([n_1]^* \times [n_2]^* \times \ldots \times [n_k]^*\big) = \prod_{i=1}^{k} (n_i + 2).$$

6. A DNF FSM WITH "TRIGGER MEMORY"

It turns out that in our application, the third of the simplifying assumptions does not hold. Recall that this was:

> (3) Per item to be analyzed, every trigger gets only set once to 0 or 1, there are no corrections or repetitions.

So, if triggers can be set more than once, we need some kind of "memory" to store the information which trigger has been set to what truth value, and which triggers are still undefined. (In the AA standup, we referred to that as the "idempotence property".)

Not surprisingly the state space (set of possible states) will increase considerably.

In this section, we build a finite state machine that stores the state of the trigger and therefore is not changed in its state if a trigger is set more than once.

We are using the same setting, variables, and indices as in section 5.

Now we give a formal description of the FSM. Outlook: it will have

- a large state space S unfortunately (but we warned of this before), and
- some good news: a very simple transition function $\delta : S \times \Sigma \to S$!

6.1. **Input alphabet Σ.** Let $N = \max\{n_1, \ldots, n_k\}$ be the maximum number of triggers that any `AND`-gate has. We set

$$\Sigma = \{1, \ldots, k\} \times \{1, \ldots, N\} \times \{0, 1\}.$$

In the last set of the Cartesian product, we interpret 0 as FALSE and 1 as TRUE.

We interpret $(i, j, b) \in \Sigma$ as

> "in `AND`-gate number i set trigger number j to b."

This begs the question, what happens if $j > n_i$, that is j is larger than the number of triggers that `AND`-gate number i has? We set the transition function $\delta : S \times \Sigma \to S$ to just do nothing in that case, that is to output the same state again. We'll come to that in a moment.

6.2. **State space S.** Our basic units of information now are

- 0, corresponding to FALSE
- 1, corresponding to TRUE
- u, corresponding to "undefined" (meaning a trigger has not yet been set).

Set

$$S := \{0, 1, u\}^{n_1} \times \{0, 1, u\}^{n_2} \times \ldots \times \{0, 1, u\}^{n_k}.$$

The set S represents all possible configurations of the triggers for each `AND`-gates. (Recall that `AND`-gate number i has n_i triggers feeding into it.)

So $v \in S$ is a vector of k elements, and for $i \in \{1, \ldots, k\}$, the element v_i is itself a vector of n_i elements, each which is either $0, 1$, or u.

Example. Let $k = 3$ and $n_1 = 2, n_2 = 5, n_3 = 3$. So we have 3 `AND`-gates, the first with 2 triggers, the second with 5, and the last with 3 triggers. One example member of S would be

$$v := \big((1, u), (u, u, 0, 0, 1), (u, u, 0)\big).$$

So for instance we have $(v_1)_2 = u$, as the second element of the first vector is u, and $(v_2)_3 = 0$.

6.3. **Set of terminal states F.** Based on the classification lemma 5.1 we can write $F := F_{\text{false}} \cup F_{\text{true}}$ where

(1) F_{false} consists of all members of S such that for all $i \in \{1, \ldots, k\}$ the vector v_i contains some element that is 0, and
(2) F_{true} consists of all members of S such that there is $i \in \{1, \ldots k\}$ such that the vector v_i is the constant-1-vector (more formally,

$$(v_i)_j = 1 \text{ for all } j \in \{1, \ldots, n_i\}.$$

6.4. **Initial state $s_0 \in S$.** We set s_0 to be the vector

$$\big(\underbrace{(u, \ldots, u)}_{\text{length } n_1}, \underbrace{(u, \ldots, u)}_{\text{length } n_2}, \ldots \underbrace{(u, \ldots, u)}_{\text{length } n_k}\big).$$

Or more formally set $(v_i)_j = u$ for all $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, n_i\}$ and set $s_0 := v$.

6.5. **Transition function $\delta : S \times \Sigma \to S$.**

- End state behaviour: for $e \in F, \sigma \in \Sigma$ we set $\delta(e, \sigma) = e$.
- Suppose $v \in S \setminus F$ and let $\sigma = (i, j, b) \in \Sigma$, where $i \in \{1, \ldots, k\}, j \in \{1, \ldots, N\}$ [1], and $b \in \{0, 1\}$. We distinguish two cases:
  - $j > n_i$: then do nothing, i.e. $\delta(v, (i, j, b)) = v$.
  - $j \leq n_i$: set the value of $(v_i)_j$ to b, and otherwise leave v unchanged. More formally, define $v' \in S$ by:

[1](recall $N = \max\{n_1, \ldots, n_k\}$)

$(v'_i)_j := b$,
$(v'_i)_y := (v_i)_y$ for $y \in \{1, \dots n_i\} \setminus \{j\}$, and
$v'_z = v_z$ for $z \in \{1, \dots, k\} \setminus \{i\}$.

Then set $\delta(v, (i, j, b)) := v'$.

Note that this finite state machine not only has a trigger memory, (so it has the idempotence property that we discussed in the AA meeting), it even allows for "corrections", that is triggers that receive input TRUE first and then get corrected to FALSE (this was not in the requirements).

The state space size of this FSM is

$$\mathrm{card}(S) = 3^{(\sum_{i=1}^{k} n_i)}.$$

(Note that $\sum_{i=1}^{k} n_i$ is the total number of triggers.)

7. An "order-theoretical" approach

7.1. **Informal description of the idea.** Suppose we are given a Boolean function

$$f : \{0,1\}^n \to \{0,1\}$$

that is entirely made up of $\wedge$ and $\vee$ and does not use negation ($\neg$). Assume that we are given the truth value (0 or 1) for some of the n triggers, and for others, we do not know their truth value. Then we propose the following procedure:

- Set all triggers with unknown truth value to 0 (FALSE) and calculate $f()$.
- Set all triggers with unknown truth value to 1 (TRUE) and calculate $f()$.

If the output values of f calculated above **agree**, we can conclude (with an argument given in the next section) that f will give the same output **no matter** what truth values we give the triggers with unknown truth value. Therefore we can give an output with confidence in that case.

If the two values do not agree, we cannot give an output for f given the partial information of the triggers of which we know the truth value.

7.2. **(Order-theoretical) justification of the method given above.**

Definition 7.1. *A* partially ordered set *or* poset *for short is a pair* $(P, \leq_P)$ *consisting of a base set* P *and a binary relation* $\leq_R \subseteq P \times P$ *such that*

(1) $(p,p) \in \leq_P$ *for all* $p \in P$ *("reflexivity");*
(2) $(p,q),(q,p) \in \leq_P$ *implies* $p = q$ *("anti-symmetry"), and*
(3) $(p,q),(q,r) \in \leq_P$ *implies* $(p,r) \in \leq_P$ *("transitivity").*

Usually we write $p \leq_P q$ *instead of* $(p,q) \in \leq_P$.

As an easy example of a poset, consider $\{0,1\}$ with the relation

$$\leq = \{(0,0),(0,1),(1,1)\}.$$

For short, we say $\{0,1\}$ is *ordered by* $0 \leq 1$.

The next example will give a hint of why these definitions will be important later: Let $P = \{0,1\}^n$ the set of all $\{0,1\}$-vectors of length n, and we say that for $x, y \in \{0,1\}^n$ we have

$$x \leq y \text{ if and only if } x_i \leq y_i \text{ for all } i \in \{1,\ldots,n\}.$$

It is an easy exercise to verify that this gives us a poset.

Definition 7.2. *Let* $(P, \leq_P), (Q, \leq_Q)$ *be posets. A map* $f : P \to Q$ *is said to be* monotone *if*

$$x \leq_P y \implies f(x) \leq_Q f(y) \text{ for all } x, y \in P.$$

The following are examples of monotone functions:

- $\wedge : \{0,1\}^2 \to \{0,1\}$,
- $\vee : \{0,1\}^2 \to \{0,1\}$.

The next lemma deals with the composition of monotone functions and is quite easy to prove.

Lemma 7.3. *If* P, Q, Z *are posets and* $f : P \to Q, g : Q \to Z$ *are monotone functions, then so is*

$$g \circ f : P \to Z.$$

So in particular, we have:

Corollary 7.4. *Any Boolean function* $f : \{0,1\} \to \{0,1\}$ *not containing negation* ($\neg$)*, but only consisting of* $\wedge, \vee$ *is a composition of the monotone functions* $\vee$ *and* $\wedge$ *(respectively, possibly multiple copies of these) and it is therefore* monotone.

We can apply the following "sandwich lemma" directly to our situation.

Lemma 7.5. *If* P, Q *are posets and* $f : P \to Q$ *is monotone, and* $a < b \in P$ *such that* $f(a) = f(b)$*, then:*

for all $x \in P$ with $a \leq x \leq b$ we have $f(a) = f(x) = f(b)$.

Let's revisit our starting situation. We are given a Boolean function $f : \{0,1\} \to \{0,1\}$ not containing negation ($\neg$), but only consisting of $\wedge, \vee$. By lemma 7.4 we know that f is monotone.

Now, we are ready to use the sandich lemma 7.5. We know the truth values for some of the inputs. If we set all other inputs to 0, we get variable a in the sandwich lemma: we go as low as possible. If we set all inputs with unknown truth value to 1, we go as high as possible, that is, we get variable b in the sandwich lemma.

Any assignment of $\{0,1\}$ to the inputs (triggers) with unknown truth value will result in an input vector x that is **sandwiched** between a and b, i.e.

$$a \leq x \leq b.$$

So we meet the pre-condition of the sandwich lemma and get that no matter what the input vector x looks like, we get $f(x) = f(a) = f(b)$.

So we can give a definite output for f even for our partial input.