Formal Methods Notes

Clemens Koza, 01126573

Contents

License

Revision history

- no “published” version yet.

About this document

This document is based on the Formal Methods in Informatics lecture at Vienna University of Technology; particularly from taking it in the 2023/24 winter term. Corrections and additions are welcome as pull requests at

https://github.com/SillyFreak/tu-wien-software-engineering-notes

This document leaves out several details and was written primarily for me, but I hope it is useful for other people as well. Among the things one reading it should keep in mind:

- This is not a replacement to the course slides or the lectures/recordings in any way. Much of the information may require context and/or preliminaries found in the slides or in the lecture.
- In general, I skip formal definitions where they were basic enough for me; for example, I do not repeat syntax and semantic definitions for SIMPLE, FOL, etc.
- The document is in general still incomplete.
- I am fallible and make errors or overlook some points’ importance.

If you have questions, feel free to reach out on Github. I may at least occasionally be motivated enough to answer questions, extend the document with explanations based on your question, or help you with adding it yourself.

1. Tseitin translation

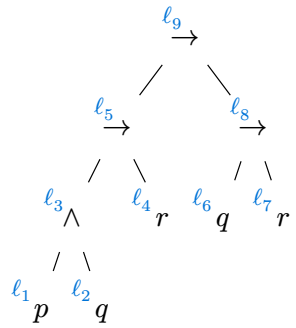
The Tseitin translation's purpose is to transform an arbitrary propositional formula into a conjunction of disjunctions (clauses), which can then be fed to a typical SAT solver. In contrast to the CNF, the resulting formula is linear in the input formula's length.

Following, we will translate the example formula from the lecture slides:

$$\varphi : (p \wedge q \to r) \to (q \to r)$$

1.1. Labelling of subformula occurrences (SFOs)

In the tree representation of the formula, every subformula receives a new atom. That new atom is set equivalent to the subformula; if the subformula is not itself atomic, the branches of the subformula are themselves replaced by new atoms.



$$\begin{aligned}
\ell_1 &\leftrightarrow p \\
\ell_2 &\leftrightarrow q \\
\ell_3 &\leftrightarrow \ell_1 \wedge \ell_2 \\
\ell_4 &\leftrightarrow r \\
\ell_5 &\leftrightarrow \ell_3 \to \ell_4 \\
\ell_6 &\leftrightarrow q \\
\ell_7 &\leftrightarrow r \\
\ell_8 &\leftrightarrow \ell_6 \to \ell_7 \\
\ell_9 &\leftrightarrow \ell_7 \to \ell_8
\end{aligned}$$

We can easily see that subformula p is satisfiable iff $(\ell_1 \leftrightarrow p) \wedge \ell_1$ is satisfiable: substituting the left clause into the right one, we are left with exactly p. This conjunction of the new atom and the corresponding equivalence clause thus captures the satisfiability of the original subformula. Likewise, for a more complex subformula like $p \wedge q$, we can take $(\ell_1 \leftrightarrow p) \wedge (\ell_2 \leftrightarrow q) \wedge (\ell_3 \leftrightarrow \ell_1 \wedge \ell_2) \wedge \ell_3$ and get the same result. Applying this to the whole formula φ, we get

$$(\ell_1 \leftrightarrow p) \wedge \ldots \wedge (\ell_9 \leftrightarrow \ell_7 \to \ell_8) \wedge \ell_9$$

This formula has three crucial properties:

- as stated above, it is satisfiable iff φ is satisfiable (although not logically equivalent because it contains new atoms),
- its length is linear in terms of the length of φ, and
- it is essentially *flattened*, which means that the linearity in length will be preserved when transforming it to CNF.

1.2. Generating clauses for the CNF

SAT solvers by convention accept formulas as a set of clauses over a set of atoms. To make the above result usable by such a tool, we need to convert it into this form. Since we have constructed a flat set of equivalence clauses, we can enumerate all possible forms of clauses and how they are translated into CNF. For example:

$$\begin{aligned}
\ell_i \leftrightarrow x &\equiv (\ell_i \to x) &\wedge (\ell_i \leftarrow x) &\equiv (\neg\ell_i \vee x) \wedge (\ell_i \vee \neg x) \\
\ell_i \leftrightarrow (x \wedge y) &\equiv (\ell_i \to (x \wedge y)) &\wedge (\ell_i \leftarrow (x \wedge y)) &\equiv (\neg\ell_i \vee x) \wedge (\neg\ell_i \vee y) \wedge (\ell_i \vee \neg x \vee \neg y) \\
\ell_i \leftrightarrow (x \to y) &\equiv (\ell_i \to (x \to y)) &\wedge (\ell_i \leftarrow (x \to y)) &\equiv (\neg\ell_i \vee \neg x \vee y) \wedge (\ell_i \vee x) \wedge (\ell_i \vee \neg y)
\end{aligned}$$

Another approach to getting these clauses' CNFs is directly via the truth table. Recall: from the true (false) entries in a truth table, one can read the DNF (CNF) of the function represented by the truth table:

x	y	$x \oplus y$	DNF	CNF
0	0	0		$x \vee \quad y$
0	1	1	$\neg x \wedge \quad y$	
1	0	1	$x \wedge \neg y$	
1	1	0		$\neg x \vee \neg y$

Each CNF clause is formed from one row where the formula is not satisfied. The first line can be read like this: "if $x \mapsto 0, y \mapsto 0$, the formula is not fulfilled, so at least one of the varibales must be opposite from that assignment." Then, if all clauses (that each negate one of the conditions under which the formula is false) are satisfied, the assignment belongs to one of the other rows, i.e. the ones that *do* satisfy the formula.

Thus, we can enumerate for each subformula in our translation the false assignments and read the CNF from that; for example ("x" stands for "don't care"):

$x \wedge y \leftrightarrow \ell$			CNF
0	x	1	$x \vee \neg \ell$
x	0	1	$y \vee \neg \ell$
1	1	0	$\neg x \vee \neg y \vee \ell$

We can now finally translate our formula into a *definitional form* that is in CNF:

$$(\neg \ell_1 \vee p) \wedge (\ell_1 \vee \neg p) \wedge \ldots \wedge (\neg \ell_9 \vee \neg \ell_7 \vee \ell_8) \wedge (\ell_9 \vee \ell_7) \wedge (\ell_9 \vee \neg \ell_8) \wedge \ell_9$$

Or, as a set of clauses instead of a formula:

$$\delta(\varphi) = \hat{\delta}(\varphi) \cup \{\ell_9\} = \{\neg \ell_1 \vee p, \ell_1 \vee \neg p, \ldots, \neg \ell_9 \vee \neg \ell_7 \vee \ell_8, \ell_9 \vee \ell_7, \ell_9 \vee \neg \ell_8, \ell_9\}$$

2. Implication Graphs

Implication graphs capture aspects of the logical structure of a set of clauses that are useful for efficiently computing satisfiability of that set of clauses. We will first define some preliminaries, then implication graphs themselves, and then look at some examples.

2.1. Clause states

A clause can have one of four states:

- *satisfied:* at least one of its literals is assigned true
- *unsatisfied:* all its literals is assigned false
- *unit:* all but one literals is assigned false (i.e. the last needs to be assigned true to be satisfied, the decision is forced)
- *unresolved:* none of the above

2.2. Decisions

Decisions add variable assignments to a partial variable assignment. Depending on when in the process a variable is assigned, a decision has a *decision level*. By deciding one variable per level, decision levels (for regular decisions) range from 1 at most to the number of variables - usually, it will be less, because decisions on unit clauses are forced.

We write $x = v@d$ to say that variable x has been assigned value v at decision level d. As a shorthand for $x = 1@d / x = 0@d$, we write $x@d / \neg x@d$ (or $\ell@d$: deciding a literal at depth d, where the literal is a negated or non-negated variable). The *dual* ℓ^d (or $\ell^d@d$) means the opposite decision of ℓ (or $\ell@d$), e.g. the dual of $x@d$ is $\neg x@d$ and vice-versa.

2.3. Antecedents

Under a partial variable assignment σ, a clause C may simplify to a unit clause ℓ, which we write $C\sigma = \ell$. For example, under the partial variable assignment $\sigma = \{x_1 \mapsto 1, x_4 \mapsto 0\}$, the clause $C : \neg x_1 \vee x_4 \vee \neg x_3$ simplifies to $\neg x_3$.

The clause that, under some σ, forces a decision of ℓ is called its *antecedent*: $\text{Antecedent}(\ell) = C$. Here, C is treated as a set of literals (i.e. $\{\neg x_1, x_4, \neg x_3\}$). Although it should not matter for the construction of implication graphs, when a literal is not forced under some partial assignment, we can simply take the antecedent to be an empty set.

2.4. The implication graph

An implication graph (IG) is a DAG $G = (V, E)$ that satisfies:

- Each vertex has a label of the form $\ell@d$, where ℓ is some literal and d is a decision level.
- For each vertex v_j, take the set of dual literals of its antecedent: $\{v_i \mid v_i^d \in \text{Antecedent}(v_j)\}$. For all v_i that are vertices of the graph, there is an edge from v_i to v_j, or: $E = \{(v_i, v_j) \mid v_i, v_j \in V, v_i^d \in \text{Antecedent}(v_j)\}$. All these edges are labelled with $\text{Antecedent}(v_j)$.
  - The way IGs are constructed, all but one v_i *will* label a vertex in the graph, or equally: the clause $\text{Antecedent}(v_j)$ is unit and the edges completely decribe why v_j was forced; unforced literals don't have incoming edges. The only v_i that thus didn't label a vertex is v_j^d. If that existed, this leads to the other option:

Conflict graphs are also implication graphs, and they contain additionally

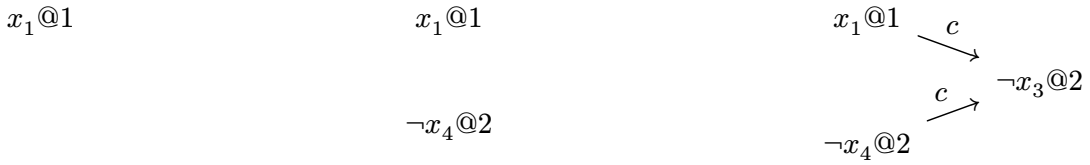
- one vertex labelled κ called the conflict node, and
- edges $\{(v, \kappa) \mid v^d \in c\}$ for some clause c.
  - Note that there's no "filter" $v \in V$ in this definition, so all of c's literals' duals need to actually be vertices, which makes the clause actually unsatisfied and κ's predecessor nodes conflicting.

Implication graphs are created incrementally by deciding one variable, resolving any unit clauses resulting from that assignment (by boolean constraint propagation or BCP), and repeating that until either all variables are assigned or a conflict is found.

2.5. Example 1

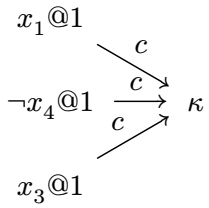
Let's take the single clause $c : \neg x_1 \vee x_4 \vee \neg x_3$ again and start from zero.

- Arbitrarily we begin by assigning $x_1 \mapsto 1$, or with the decision $x_1@1$.
- After this decision, we have no unit clause and continue with $x_4 \mapsto 0$ or $\neg x_4@2$. At this point, c can be simplified to $\neg x_3$, meaning we have $\text{Antecedent}(\neg x_3) = \{\neg x_1, x_4, \neg x_3\}$.
- We thus add a vertex $\neg x_3@2$ (at the same depth as the decision that made the clause unit) and two edges $(x_1@1, \neg x_3@2)$, $(\neg x_4@2, \neg x_3@2)$ to the graph.
- All variables have been assigned without conflict; we're done.



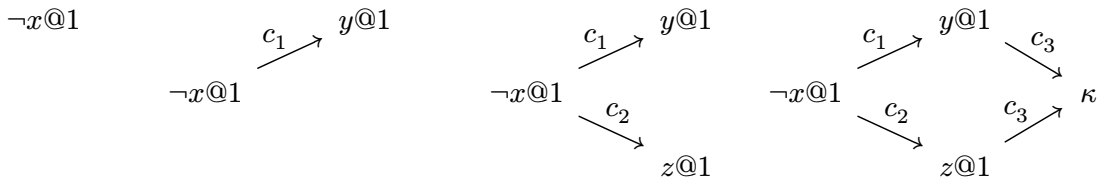
2.6. Example 2

Let's assume that we had not made decisions one after the other (and thus resolved c that became a unit clause) but came up instantly with the assignment $\{x_1@1, \neg x_4@1, x_3@1\}$. This would make c unsatisfied and lead to a conflict graph:



2.7. Example 3

To arrive at a conflict organically, we need a set of clauses that can be refuted such as $c_1 : x \vee y$, $c_2 : x \vee z$, $c_3 : \neg y \vee \neg z$. We start by assigning $x \mapsto 0$ and are forced from there:



Sneak peek into CDCL: our decision of $\neg x@1$ led to a conflict here (in general, there could be multiple unforced decisions, but here it's just one), so we know that that is a decision we *must not* make. We thus need to backtrack to before that decision (in this case: before *any* decision) and prevent ourselves from making that mistake again.

The way this is done is by adding a clause that captures the wrong decision, i.e. $c_4 : x$. Because here, we only had one unforced decision, this is already a unit clause, and we're forced to decide $x@0$ (i.e. before the first regular decision).

After that, no clause is unit (c_1, c_2, c_4 are satisfied, c_3 is unresolved) so we can make a decision such as $y@1$, which forces a decision on z:

The core of conflict-driven clause learning (CDCL) is then to define algorithms for finding conflict clauses and backtracking strategies.