Crafted Combinational Equivalence Instances

William Klieber
Carnegie Mellon University
Pittsburgh, PA, USA
{wklieber}@cmu.edu

I. Background

In formal verification, one often wants to check whether two propositional formulas or combinational logic circuits are equivalent. Given two propositional formulas ϕ_1 and ϕ_2, we can check equivalence by taking the exclusive-or (XOR) of these formulas, denoted "$\phi_1 \oplus \phi_2$", and querying whether this formula is satisfiable. We present a generator for creating problems of this nature, available at:
https://www.cs.cmu.edu/%7Ewklieber/bench-sat2017/benchgen.py

II. Overview

This benchmark suite contains two classes of benchmarks: satisfiable and unsatisfiable. The unsatisfiable formulas are created as follows. First, a propositional formula ϕ is randomly created, as detailed in section III. This formula is then refactored, as detailed in section IV, to produce a logically equivalent but syntactically very different formula ϕ'. Finally, we take the XOR of ϕ and ϕ' and encode it in DIMACS.

For the satisfiable instances, we proceed as follows. We first generate a formula ϕ, mostly in the same way as for an unsatisfiable instance, but with a slight complication explained in section V. Then, before refactoring it, we first slightly modify ("tickle") it to produce a formula that is guaranteed to not be logically equivalent, as described in section VI. Let ϕ' be the refactored tickled formula. So, $\phi \oplus \phi'$ is satisfiable. However, it turns out that modern SAT solvers can easily solve even large instances of this form. So, to make things more challenging, we instead create k formulas $(\phi_1, \ldots, \phi_k)$ for some small k. (In particular, we use $k = 12$.) We randomly generate a single assignment A and then tickle and refactor the original formulas to produce modified formulas $(\phi'_1, \ldots, \phi'_k)$ in such a way that $\phi_i|_A \neq \phi'_i|_A$ for $i \in \{1, ..., k\}$, where "$\psi|_A$" denotes the truth value that ψ evaluates to under A. The final formula is the conjunction:

$$(\phi_1 \oplus \phi'_1) \wedge \cdots \wedge (\phi_k \oplus \phi'_k)$$

III. Generation Of Random Formulas

We generate formulas with the following BNF grammar:

$$\begin{aligned}
\mathit{AndFmla} &::= \texttt{AND(}\mathit{XorFmla}\texttt{,}\ \mathit{XorFmla}\texttt{)} \ \mid\ \mathit{Lit} \\
\mathit{OrFmla} &::= \texttt{OR(}\mathit{XorFmla}\texttt{,}\ \mathit{XorFmla}\texttt{)} \ \mid\ \mathit{Lit} \\
\mathit{XorFmla} &::= \texttt{XOR(}\mathit{AndFmla}\texttt{,}\ \mathit{OrFmla}\texttt{)} \\
&\ \mid\ \ \texttt{XOR(}\mathit{OrFmla}\texttt{,}\ \mathit{AndFmla}\texttt{)} \ \mid\ \mathit{Lit} \\
\mathit{Lit} &::= \mathit{Var} \ \mid\ \neg\,\mathit{Var}
\end{aligned}$$

In other words: Each gate has two children. Each child of an AND or OR gate is either an XOR gate or a literal. Each XOR gate has one AND child and one OR child, unless one or both of these children are literals instead.

The formula, viewed as tree, is a balanced tree. In a subtree with 8 or fewer leafs, each leaf has a distinct variable. This avoids degenerate cases such as $\texttt{AND}(x, \neg x)$ and helps avoid producing such subformulas during refactoring (section IV).

IV. Refactoring Of Formulas

First all the gates of the formula are converted to *if-then-else* (ITE) gates, as follows:

$$\begin{aligned}
\texttt{AND}(x, y) &= \texttt{ITE}(x, y, \texttt{false}) \\
\texttt{OR}(x, y) &= \texttt{ITE}(x, \texttt{true}, y) \\
\texttt{XOR}(x, y) &= \texttt{ITE}(x, \neg y, y)
\end{aligned}$$

Negations are pushed inwards so that they occur only directly in front of variables. Then, some subformulas of the form

$$\texttt{ITE}(\texttt{ITE}(sel, t_{in}, f_{in}), t_{out}, f_{out})$$

are refactored to the following logically equivalent form:

$$\texttt{ITE}(sel,\ \texttt{ITE}(t_{in}, t_{out}, f_{out}), \texttt{ITE}(f_{in}, t_{out}, f_{out}))$$

Specifically, we define a recursive procedure *Refactor* as follows:

$\mathit{Refactor}\big(\texttt{ITE}(\texttt{ITE}(sel, t_{in}, f_{in}), t_{out}, f_{out})\big)$ returns either

$$\begin{aligned}
\mathit{Refactor}\big(\texttt{ITE}(sel,\ &\mathit{Refactor}(\texttt{ITE}(t_{in}, t_{out}, f_{out})), \\
&\mathit{Refactor}(\texttt{ITE}(f_{in}, t_{out}, f_{out})))\big)
\end{aligned}$$

or

$$\begin{aligned}
\texttt{ITE}\big(&\mathit{Refactor}(\texttt{ITE}(sel, \texttt{true}, \texttt{false})), \\
&\mathit{Refactor}(\texttt{ITE}(t_{in}, t_{out}, f_{out})), \\
&\mathit{Refactor}(\texttt{ITE}(f_{in}, t_{out}, f_{out}))\big)
\end{aligned}$$

with the choice of these two options determined partially at random. As the base case, $\mathit{Refactor}\big(\texttt{ITE}(lit, t_{out}, f_{out})\big) = \texttt{ITE}(lit, t_{out}, f_{out})$, where *lit* is a literal.

V. Pretickling Of Formulas

When creating satisfiable instances, there is an additional step in randomly generating a formula. After the steps in section III are completed, the formula is *pretickled* to produce a semantically different (i.e., not logically equivalent) formula that is suitable for input to the *Tickle* function described in section VI. The purpose of this is to ensure that, for a predetermined randomly generated assignment A, Tickle can flip the truth value of the formula ϕ by flipping the polarity of one of its leafs. Let L_{flip} be the leaf whose polarity we will flip. Let P be the path from the root of ϕ to L_{flip}. Then, for each gate G of the form $\texttt{AND}(x, y)$, $\texttt{AND}(y, x)$, $\texttt{OR}(y, x)$, or $\texttt{OR}(x, y)$, where G and x are on the path P (and therefore y is not), we must ensure that y does not control the output of G. If G is an AND gate and $y|_A = \texttt{false}$, then $\mathit{Pretickle}$ replaces y with its negation. Likewise, if G is an OR gate and $y|_A = \texttt{true}$, $\mathit{Pretickle}$ replaces y with its negation.

VI. Tickling Of Formulas

Given an assignment A and a formula ϕ produced by $\mathit{Pretickle}_A$, the Tickle_A function flips the polarity of a single leaf node (literal) of ϕ such that $\phi|_A \neq \mathit{Tickle}_A(\phi)|_A$. As in section V, let L_{flip} be the leaf whose polarity we will flip, and let P be the path from the root of ϕ to L_{flip}. We define the Tickle_A function as follows, where $op \in \{\texttt{AND}, \texttt{OR}, \texttt{XOR}\}$:

$$\mathit{Tickle}_A(op(x, y)) = \begin{cases} op(\mathit{Tickle}_A(x), y) & \text{if } x \text{ is on } P \\ op(x, \mathit{Tickle}_A(y)) & \text{if } y \text{ is on } P \end{cases}$$

$$\mathit{Tickle}_A(lit) = \neg lit \quad \text{for a literal } lit$$