Crafted Combinational Equivalence QBF Instances

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

I. Introduction

In formal verification, one often wants to check whether two propositional formulas or combinational logic circuits are equivalent. The QBFs in this benchmark suite address a related type of problem: "Does there exist an assignment to a set of variables V_E under which one propositional formula ϕ_1 becomes logically equivalent to another propositional formula ϕ_2?". Accordingly, the QBFs of this benchmark suite have the form:

$$\exists V_E . \forall V_A .\ \phi_1 \Leftrightarrow \phi_2$$

where V_A and V_E are two disjoint sets of boolean variables, and ϕ_1 and ϕ_2 are propositional formulas, and the set of variables in ϕ_1 is $V_A \cup V_E$, and the set of variables in ϕ_2 is V_A. We present a generator for creating such problems: https://www.cs.cmu.edu/%7Ewklieber/bench-sat2017/benchgen.py

II. Overview

This benchmark suite contains two classes of benchmarks: *equivalent* (for which the QBF evaluates to `true`) and *inequivalent* (for which the QBF evaluates to `false`). The equivalent QBFs are created as follows. First, a propositional formula ϕ_0 is randomly created, as detailed in section III. Formula ϕ_1 is then produced from ϕ_0 as detailed in section IV, so that there is at least one assignment to V_E under which ϕ_1 becomes equivalent to ϕ_0, and ideally many assignments under which it does not. Formula ϕ_2 is produced by refactoring ϕ_0, as detailed in section V, to produce a formula logically equivalent to ϕ_0 but syntactically very different. The final QBF is $\exists V_E . \forall V_A .\ \phi_1 \Leftrightarrow \phi_2$.

For the inequivalent instances, we proceed as follows. We first generate a formula ϕ_0, mostly in the same way as for an equivalent instance, but with a complication explained in section VI. Formula ϕ_1 is then created from ϕ_0 exactly as described above for equivalent instances. To generate ϕ_2 from ϕ_0, we first slightly modify ("tickle") ϕ_0 to produce a formula that is guaranteed to not be logically equivalent to ϕ_0, as described in section VII, and then refactor it as described in section V. Note that it is not guaranteed that ϕ_1 is inequivalent to ϕ_2 under every assignment to V_E. Depending on which variables from V_A were randomly selected when creating ϕ_1 from ϕ_0 in section IV, there might actually exist an assignment to V_E that makes ϕ_1 equivalent to ϕ_2. However, experimental evidence indicates that this is unlikely.

III. Generation Of Random Formulas

We generate formulas with the following BNF grammar:

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

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 V).

IV. Splitting the Leafs

The formula ϕ_0 can be viewed as tree, where each leaf node is a literal. We create ϕ_1 from ϕ_0 by replacing each leaf node ℓ with a formula of the form $\texttt{ITE}(e, \ell, u)$, where e is a variable from V_E and u is a randomly selected literal such that u or $\neg u$ is in V_A. "$\texttt{ITE}(x, y, z)$" denotes an *if-then-else* gate; it is logically equivalent to $(x \wedge y) \vee (\neg x \wedge z)$. A different existential variable is used for each leaf node of ϕ_0.

V. Refactoring Of Formulas

First all the gates of the formula are converted to `ITE` gates:

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

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:

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

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

or

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

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

VI. Pretickling Of Formulas

When creating inequivalent 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 VII. 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 *Pretickle* replaces y with its negation. Likewise, if G is an `OR` gate and $y|_A = \texttt{true}$, *Pretickle* replaces y with its negation.

Note: In an earlier version of the benchmark generator, a different algorithm was used for the *Pretickle* function. In particular, the old version of *Pretickle* examined every gate in the formula instead of only gates along a path. If both inputs to an `AND` gate evaluate to `false` under A (or both inputs to an `OR` gate evaluate to `true` under A), the old version of *Pretickle* would replace one of these inputs with its negation. The command-line argument "`--pretickle-path`" controls which version of *Pretickle* is used.

VII. 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 VI, 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 $\mathit{op} \in \{\texttt{AND}, \texttt{OR}, \texttt{XOR}\}$:

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

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

VIII. Summary

In summary, we produce a QBF of the form $\exists V_E . \forall V_A .\ \phi_1 \Leftrightarrow \phi_2$ where ϕ_1 and ϕ_2 are produced as follows:

```
if (is_equivalent_instance) {
    phi_0 := random_fmla();
    phi_1 := split(phi_0); /* section IV */
    phi_2 := refactor(phi_0);
} else {
    A := random_assignment();
    phi_0 := pretickle(A, random_fmla());
    phi_1 := split(phi_0); /* section IV */
    phi_2 := refactor(tickle(A, phi_0));
}
```