1 Partial order reduction

Partial order reduction (POR) is a class of algorithms that can be used to reduce the number of states explored when model checking a distributed system. Distributed systems have a unique problem when model checking: each possible interleaving of each operation by each process must be examined, unless it can be proved that some interleavings are equivalent. POR techniques can reduce the size of the state space by eliminating some of these equivalent paths.

One possible way to perform POR is to calculate a set $ample(s)$ for each state s. $ample(s)$ contains the transitions that are sufficient to explore, with the condition that the reduced state space is equivalent to the full one. $ample(s)$ should be as small as possible.

$ample(s)$ must satisfy four conditions:

- C0 Emptiness: The $ample(s)$ set should not be empty unless the $enabled(s)$ set is empty. Search algorithms must be able to make progress on the reduced state space if they would be able to on the full state space.
- C1 Ample decomposition: A transition dependent on a transition from $ample(s)$ cannot appear before some transition from $ample(s)$ is executed. Dependency relationships between transitions must be maintained.
- C2 Invisibility: If $ample(s) \neq enabled(s)$, every transition in $ample(s)$ must be invisible.
- C3 Cycle closing condition.

There is no efficient algorithm for calculating $ample(s)$ in the general case. In particular, "in general checking C1 is at least as hard as checking reachability for the full state transition graph." Practical POR algorithms exploit properties of their problem domain to calculate $ample(s)$.

The SyGuS problem can be modeled as a state space search on the space of expression trees of a grammar G. Each state consists of a partial expression tree. Each transition corresponds to an expansion of a non-terminal symbol according to G. The terminal states are those that do not contain any non-terminal symbols; these are complete expressions. In the general case, the state space could contain cycles. However, disallowing grammars that can produce cycles is not a major restriction. Many sequences of grammar expansions are equivalent, and could be eliminated by POR. For example, for commutative operators, only one ordering of each combination of children should be examined.

The SyGuS problem has some challenges that make applying POR difficult. Its state space is infinite, so a breadth-first search must be used. Most POR algorithms use a depth first search (including dynamic partial order reduction). Eliminating expressions that are equivalent due to commutativity is easy, but avoiding other kinds of equivalence is not. The normalizing CEGIS algorithm below performs the same task, but is more flexible.

2 Enumerative CEGIS

The enumerative CEGIS algorithm from "TRANSIT: Specifying Protocols with Concolic Snippets" does not use POR. It uses a basic CEGIS loop: it finds a candidate expression, checks it using the solver, and uses the counterexample model to refine the search.

2.1 Expression generation

Expressions are generated in order of size, where size is the number symbols in the expression. Expressions are generated bottom-up, by applying each operator to all combinations of smaller expressions, in order to expression size.

When a new expression is generated, a "signature" is calculated. The signature is a tuple of the results of evaluating the expression under each of the counterexample models obtained during the search. In order to prune the search space, expressions with identical signatures are assumed to be equivalent. If an expression is enumerated that has a signature equal to what is already stored, it is discarded. This technique has the benefit of pruning the search space dramatically, but it requires expression enumeration to be restarted every time a new counterexample is generated.

2.2 Search

Each expression signature is compared against a goal signature, which is the tuple of the desired results for each counterexample. If the two signatures match, the expression is sent to the solver. If the expression satisfies all constraints, then the search is over. Otherwise, the solver will return a model and the expression generation phase is restarted from the beginning. This process repeats until a solution is found, or indefinitely if a solution does not exist.

3 Normalizing CEGIS

The normalizing search is a variant of the enumerative CEGIS algorithm. The enumerative algorithm uses a pruning strategy that is designed to be agressive but not accurate. The search must restart whenever the pruning is found to be incorrect. The normalizing algorithm uses a more conservative pruning strategy which only removes expressions that are provably equivalent. This results in a larger search space but means that the search never needs to be restarted.

To prune the search space, the normalizing algorithm converts each expression into a normal form before storing it. Any expression that contains a commutative or associative operator is a mamber of an equivalence class. For example, $(a + b) + c = (b + a) + c = a + (b + c)$. These three expressions all normalize to `(+ a b c)`. An expression in normal form can be denormalized to obtain a member of its equivalence class. As an added optimization, the normalizing process also folds constants. I may add a term rewriting step to handle algebraic identities as well.

My timing results show that the normalizing algorithm is competitive on small examples but is increasingly slower as the size of the search space increases.

Algorithm 1 Expression tree normalization

Let T be an expression tree.
Let $Assoc$ be the set of associative operators in T.
Let $Commut$ be the set of commutative operators in T.
function NORMALIZE($T, Assoc, Commut$)
    if $T.operator.arity = 0$ **then**
        return T
    if $T.operator \in Assoc$ **then**
        $children \leftarrow []$
        for $child \in T.children$ **do**
            if $child.operator = T.operator$ **then**
                $children \leftarrow children +$ NORMALIZE($child, Assoc, Commut$).$children$
            else
                $children \leftarrow children + [child]$
    else
        $children \leftarrow$ [NORMALIZE($c, Assoc, Commut$) | $c \in T.children$]
    if $T.operator \in Commut$ **then**
        $T.children \leftarrow$ SORT($children$)
    else
        $T.children \leftarrow children$
    return T

Algorithm 2 Expression tree denormalization

Let T be a normalized expression tree.
function DENORMALIZE(T)
    if $T.operator.arity = 0$ **then**
        return T
    if $|T.children| > T.operator.arity$ **then**
        Create a new tree T_c with the same operator as T
        $T_c.children \leftarrow T.children[T.operator.arity - 1 :]$
        $T.children$ = [DENORMALIZE(c) | $c \in T.children[: T.operator.arity - 1]] +$ [DENORMALIZE(T_c)]
    else
        $T.children \leftarrow$ [DENORMALIZE(c) | $c \in T.children$]
    return T

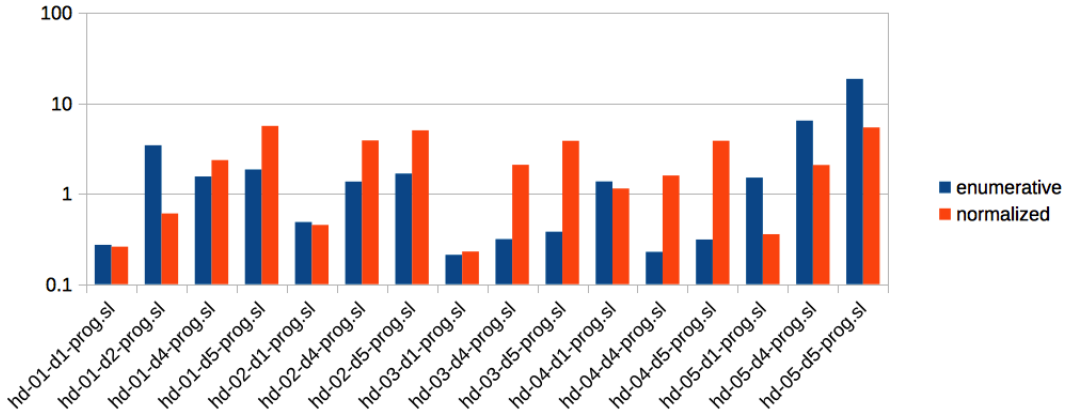

Figure 1: Timing results (sec) for benchmarks 1-5.

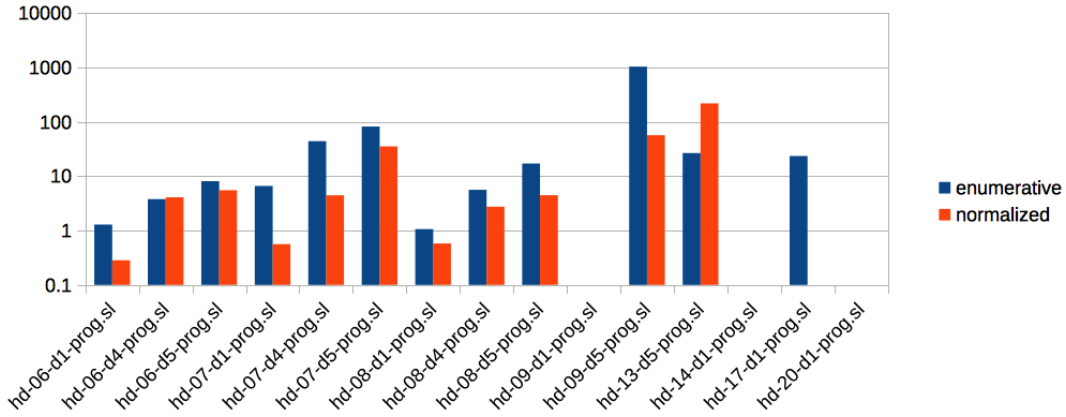

Figure 2: Timing results (sec) for benchmarks 6-20.