Using MUS for SMART++

SMART generates new assertions in blocks. Assertions are represented by the set, A. Blocks are provided with some varible set that is a subset of all variables present in the traces.

$$\hat{V} \subseteq V$$

Each block contains a number of threads that recived a subset of $\hat{V}$ and produces a single assertion. Currently $\hat{V}$ is generated randomly from V. The idea is to use the Minimal Universal Subset, MUS, to guide in the varibales to be included in future $\hat{V}$s.

Preliminaries

MUS and MSA

Exhaustative methods of finding the MUS are expensive, so we propose an approximate method that returns a set, N, for which every variable in N is underspecified, formall;

$$N = \{\, v \in V \mid \exists (V \setminus N) \,.\, \forall N \,.\, A \,\} \tag{1}$$

There exists some assignments to all varaibles not in N such that A is true for any assignment to varibales in N, suggesting that variables in N are underspecified. This is not MUS as it makes no claim to N being minimal. Note that the MUS is the dual of Minimal Satisfying Assignments, MSA, a minimal parital assignment of variables in A such that A is always true.

$$MSA = (M, \sigma).A$$

where (M, σ) is a partial assignemnt to V, M is a model and σ is a mapping from variables to the assigment.

MHS

A Minimal Hitting Set (MHS) of a A is a minimal subset of V such that for each assertion, $a \in A$, some variable of a is in the MHS.

$$MHS = \{T \subseteq V \mid \forall a \in A \,.\, T \cap a \,.\, \neq \emptyset\}$$

We can generate an approximate MHS for A as follows;

```
def approx_MHS(AV, V, T={}):
"""
 AV: List of sets of variables in each assertion in A
 T:  A potentially non-empty set of variables we think we want to include
"""
  uncovered = V
  sorted_vars = vars_by_count(AV)
  while is_uncovered(AV):
    next_v = sorted_vars.pop(0)
    T.add(next_v)
    uncovered = {x for x in uncovered if next_v not in x}
  return T
```

We will say that an (approx) MHS covers A if $V \setminus MHS$ is an MUS (the MHS is an approx MSA).

Algorithm

Give a set A of assertions, the algorithm to get an approximate MUS proceeds as follows:

We search for an approx MSA by starting with an approximate MHS . This is a good starting point, given the each assertion in A is true, we know that our MSA must hit each assertion with at least one varaible.

Partial assignemnt to the the variables in the MHS is usually not sufficient to ensure A is true for all assignemnts to variables in $V \setminus MHS$ and hence we need to add variables to the MHS until it covers A. This is described in the following pseudocode:

```
def search(AV, V, A, its):
  mhss = [approx_MHS(AV)]
  msa_sets = []
  for _ in range(its):
    mhs = mhss.pop()
    if is_mus(V/mhs, A):
          # Optional: decend_to_boundary(mhs, V, A)
      msa_sets.append(mhs)
    else:
      msa = ascend_to_boundary(mhs, V, A)
      msa_sets.append(msa)
      # initialise new mhs by starting with unsatisfiable vars from previous candidate
      mhss.append(approx_mhs(AV, mhs—msa))
  return msa_sets
```

To check whether $V \setminus MHS$ is an MUS (and hence MHS is an MSA) we use the query described in equation 1 by calling to an SMT sovler (usually CVC5). We make this more efficient by storing previously tested candidates, C. Given some new MUS query, q there are three possibilities:

1. If $\exists c \in \{c \in C \mid \text{ismus}(c)\} \mid q \subseteq c$ then q is also MUS
2. If $\exists c \in \{c \in C \mid \neg\text{ismus}(c)\} \mid c \subseteq q$ then q is not an MUS
3. Otherwise; test using CVC5 and store in C as appropriate

If the MHS is not an MSA we introduce more variables from V until it covers A. To be more targetted with how we add variables, we attempt to extract an unsat core from our MUS query, q, and add those varaibles to the MHS. However, querying with quantifiers stops us being able to automatically extracting this from CVC5. Instead we overapproximate the unsat core by checking for each assertion in A, if the relavent variables in q are not underspecified.

```
def approx_unsatcore(q, A):  # q is a set of, potentially MUS,  variables
  specified = {}
  for a in A:
    q_p = {v for v in a if a in q}
    if q_p.issubet(specified):
      continue
    if not is_mus(q_p, a):
      specified = specified.union(q_p)
```

```
  return specified
```

This is used in `ascend_to_boundary` to get an approximate MSA:

```
def ascend_to_boundary(MSA, V, A):
  specified_vars = approx_unsatcore(V/MSA)
  return MSA.union(specified_vars)
```

The final idea is to use the `unsat_vars` to guide the generation of the next MHS that we propose as a candidate MSA. We know that this set of variables need to be specified for some subset of the assertions. However, this is an overapproximation of the variables as there may be redundency between them, given that we check an assertion at a time not the whole set A together, and that these variables were unsat given the specific MHS.

Doing so may arise in scenarios where our MHS is an MSA, including redundant variables. For this we can, optionally, `descend_to_boundary`; where we greedily remove each variable until we reach the boundary where the candidate is no longer an MSA.

```
def descend_to_boundary(q, V, A):
  for v in q:
    q_p = q/{v}
    if not is_mus(V/q_p, A)
      return q
```

More traces

When we find a valid MUS we get a subset, N, of V that is underspecified, but we also get an assignment for $V \setminus N$ that models the hardware, given A. The idea is to use this assignment and test if it is a positive or negative trace;

1. If it is positive, add it to the positive traces and continue
2. If it is negative, add it to the negative traces and re-run the block with the same variable set, $\hat{V}$, given this new negative trace.

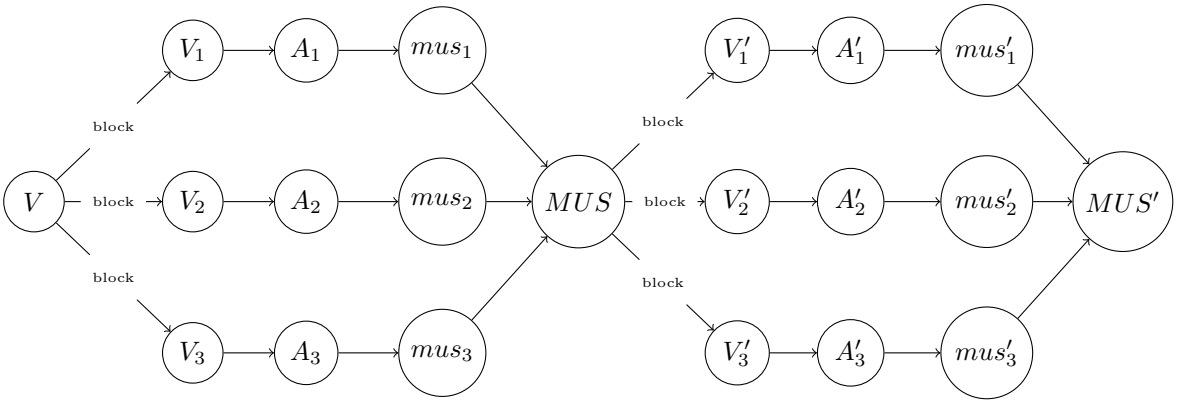
This should further ensure we are generating good asseritons, increasing mutation detection.

Engineering

This method works but is compute heavy for large A. This is due to the SMT query and the large search space when ascending/descending. Thus to effectively integrate this into SMART++ we can use the following structure;

1. Generate assertions for all variables in V
   - Split V into n independent sets, each sent to a block
   - This returns an independent set of assertions A_i
   - For each A_i we generate mus_i
   - After n blocks we generate MUS from $\bigcup_{i=1}^{n} mus_i$
2. Given all variables are covered in $\bigcup_{i=1}^{n} A_i$
   - Split MUS into n independent sets, each sent to a block
   - continue as above

This improves the efficiency of MUS calculation as we can run it on subsets of A. And we are still able to capture variables dependecies through iterative MUS generation and refinement.


V
block
block
block
V_1
V_2
V_3
A_1
A_2
A_3
mus_1
mus_2
mus_3
MUS
block
block
block
V_1'
V_2'
V_3'
A_1'
A_2'
A_3'
mus_1'
mus_2'
mus_3'
MUS'