Using MUS for SMART++

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

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

Each block contains a number of threads that recived a subset of $\hat{V}$ and produce a single assertion. Currently $\hat{V}$ is generated randomly from V. The idea is to use the Minimal Universal Subset ,MUS, to provide some guidence in the varibales to be included in $\hat{V}$. This should both speed up SMART++ and increase mutation detection and provides strengths the story of SMART++ being an effective parallelisation of specification mining through CEGIS based synthesis.

Preliminaries

MUS and MSA

Exhaustative methods of finding the MUS is expensive, so we propose an approximate method that returns a set, N, for which every variable in N is universally quantified, formally,

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

This says, that there exists some assignments to all varaibles not in N such that A is true for any values given to varibales in N, suggesting that variables in N are underspecified. But makes no claim that N is minimal. Note that the MUS is the dual of the Minimal Satisfying Assignment, MSA, a minimal parital assignment of variables in A such that A is true.

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

where (M, σ) is a partial assignemnt to V.

MHS

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

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

We can generate an approximate MHS for A as follows;

```
def approx_MHS(AV, 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
"""
  T = T or {}
  for s in AV:
    if not any(v in T for v in s):  # s not hit by T?
          v =  s[0]  # get first variable in s
          T = T.union(v)
  return T
```

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

Algorithm

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

We search for a set of approx MSA by starting with an approximate MHS. This is the case, since given that we know 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 to add variables to the MHS until it is sufficient to cover the truth of A. This search proceeds as 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 an $V \setminus MHS$ is an MUS (and hence MHS is an MSA) we use the query 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 underspecified
2. If $\exists c \in \{c \in C \mid \neg\text{ismus}(c)\} \mid c \subseteq q$ then q is not underspecified
3. Otherwise; test using CVC5 and store in C as mus or not

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 these 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 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):
  unsat_vars = approx_unsatcore(V/MSA)
  return MSA/unsat_vars
```

Need to check whether removing every `unsat_var` defintely provides a valid MSA?

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 given set of candiadte. 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.

Doing so may arise in scenarios where our MHS is an MSA, including reduendant variables. For this we can (optional) `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

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++