Week_6 - README

Draft 2

This is a tool that automatically synthesizes linear loop invariants for Dafny programs using template-based enumeration and Z3 constraint solving.

High-level idea

The tool discovers loop invariants of the form $a_1*x_1 + a_2*x_2 + \dots + a_n*x_n \le c$ by:

1. **Parsing loops** from Dafny source code (both `while` and `for` loops)
2. **Building transition relations** from loop body assignments
3. **Enumerating candidate invariants** using small integer coefficients
4. **Verifying inductiveness** via Z3: checking initialization, preservation, and satisfiability

Example

We show this tool in action via the following example -

Input (Counter method):

```
method Counter(n: int) returns (i: int)
{
  i := 0;
  while (i < n) {
    i := i + 1;
  }
}
```

Output:

```
{
 "method_name": "Counter",
 "loops": [{
   "condition": "i < n",
   "variables": ["i"],
   "synthesized_invariants": ["-i <= 0"]
 }]
}
```

The tool discovered `i >= 0` (equivalent to `-i <= 0`), which captures that the counter remains non-negative throughout execution.

Core Algorithm

```
def check_template_is_invariant(
    """
    Check template sum(ai * xi) <= c is a valid loop invariant:
      1. Initialization: holds at approximated loop entry.
      2. Preservation: (I ∧ guard ∧ T) ⇒ I'.
      3. Termination: I is satisfiable.
    """
```

The tool verifies three properties for each candidate:

- **Initialization**: The invariant holds with initial variable values
- **Preservation**: If the invariant holds before a loop iteration (with guard true), it holds after
- **Satisfiability**: The invariant is not trivially false (That is, we don’t want nonsense like x<0 and x>10.)

The tool outputs synthesized invariants for each loop in JSON format, pruning redundant invariants that are logically implied by others.

Implementation details

Loop Detection

This is supported via two strategies

- **Week 5 parser**: Uses pre-parsed loop metadata with condition and variables
- **Regex-based extraction** (for when summaries are unavailable)

```
WHILE_RE = re.compile(r"\bwhile\s*\((?P<cond>[^)]*)\)")
FOR_RE = re.compile(
    r"\bfor\s+(?P<var>" + IDENT + r")\s*:=\s*(?P<start>[^\s]+)\s*to\s*(?P<end>[^\s{]+)"
)
```

Transition system construction

Assignments are parsed and converted to Z3 constraints:

```
def build_transition_constraints(assigns, vars_before, vars_after):
    # For each 'x := rhs', encode: vars_after[x] = rhs(vars_before)# Unchanged variables: vars_after[v] = vars_before[v]
```

Only **linear expressions** (integer constants, variables, `+`, `-`) are supported. Non-linear assignments leave variables unconstrained.

Template Enumeration

Candidates are generated with (configurable) coefficient ranges:

```
def candidate_linear_templates(
    """
    generate templates of form sum(ai * xi) <= c with small integer coeffs.
    """
```

The default ranges are [-2,2] for the coefficients, and [-5,5] for the constants.

Pruning the invariants

We reduce the output size in two ways

1. **Normalization**: Divide by GCD

```
normalize_coeffs((2, 4, 6), 8) → ((1, 2, 3), 4)
```

2. **Implication pruning**: Remove invariants that are implied by other stronger ones

```
    # If "i <= n" implies "2*i <= 2*n", keep only "i <= n"
prune_implied(invariant_map)
```

The function max_invariants returns (at most) 5 synthesized invariants.