Quantifier Elimination INvariant Generator

> Equations included in documentation but not rendered in GitHub README. Please see pdf version here

This tool is designed to parse simple imperative functions containing loops and use the method of quantifier elimination to automatically generate loop invariants. To read more about the method of quantifier elimination, please read Professor Kapur's paper .

Links to Code Modules

- Simple Imperative Langue Grammar
- Parser
- Solver

Since the tool is configured as a web application and intended to eventually be deployed on a server to promote accessibility, some additional environment setup is required. The backend (including the invariant generator) runs in Python 3.7 using Django>2.2, and additional package requirements can be found in the requirements.txt file. The frontend (including the function parser) runs in NodeJS and requires npm and several modules.

Backend Setup

Once Python 3.7 is installed, navigate to the `backend` directory and run `pip install -r requirements.txt`. To run the solver API, run `python manage.py runserver` from `\backend`. The API can be accessed at `http://localhost:8000` from your browser or an HTTP request tool.

Frontend Setup

With NodeJS installed, navigate to `/frontend` and run the command `npm install` to install the required node modules. Once installation is complete, run the user interface using `npm start` in `/frontend`, then navigate to `http://localhost:3000` in your browser.

Program Options

> *Note*: Currently, program options are applied to all loops being analyzed. The tool will be extended to allow users to set options individually for each loop appearing in the function.

Variable Selection

Once the function body is parsed, program variables are automatically detected and can be selected for inclusion in automatically generated templates via a checkbox list.

Automatic Template Generation

When selected, the tool will automatically generate a polynomial template using the variables chosen by the user up to the specified degree.

For example, if the variables selected are x and y, and the degree is chosen to be true, a template with the following terms would be generated:

$$Ax^2y^2 + Bx^2 + Cx^2y + Dy^2 + Ey^2x + Fxy + Gx + Hy + I = 0$$

The size of the template quickly expands with larger numbers of variables and higher degree. To avoid stalled computations, the maximum recommended degree is between 2 and 3 depending on the number of variables.

User Supplied Terms

The user is also able to manually supply a list of terms for inclusion in the template. These terms can be polynomial, exponential, or logarithmic, and should be supplied in python notation. The constant term is automatically added to the template and does not need to be included. For example, providing the term list “x**2, 2**y, x*log(y)” would result in the following template:

$$Ax^2 + B2^y + Cx \cdot log(y) + D = 0$$

> *Note*: Functionality will be expanded to allow the user to select an automatically generated polynomial template and supplement it with additional user-supplied logarithmic or exponential terms.

Computing Invariants

Example Function

```
function example(x){
    s = 0;
    n = 1;
    while(x>0){
        s = s+n;
        n = n+2;
        x = x-1;
        }
}
```

Parsing the Function Body

To extract information from the supplied function, a parser was written using a grammar for a simple imperative language designed with ANTLR,

The parser extracts function arguments (constants), assignment statements, loops and their associated conditions (including nested loops), and conditional statements. The parse tree is used to generate a set of verification conditions corresponding to locations in the function code – these locations include after initial assignments, at each branch of a conditional statement, and when the code enters and exits a loop.

Example Verification Condition

$$T = 0 \implies T|_{s,n,x}^{<s+n,\ n+2,\ x+1>} = 0$$

Output JSON

```
{
    "left": {
      "loopIndex": 1,
      "condition": "x>0",
      "substitutions": []
    },
    "right": {
      "loopIndex": 1,
      "substitutions": [
        {
          "var": "s",
          "assignment": "s+n"
        },
        {
          "var": "n",
          "assignment": "n+2"
        },
        {
          "var": "x",
          "assignment": "x-1"
        }
      ]
    }
  }
```

Solving for Constraints

The verification conditions derived by the parser are passed to a solver written using the Python symbolic computation package SymPy to generate constraints on template function parameters.

For each verification condition, the appropriate substitutions are made to the template function. The original template is then subtracted from the template with substitutions, and all terms are fully expanded.

Using the example VC,

$$T = 0 \implies T|^{<s+n,\, n+2,\, x+1>}_{<s,n,x>} = 0$$

and the template,

$$An^2 + Bn + Cs + D = 0$$

would result in the following:

$$A(n+2)^2 + B(n+2) + C(s+n) + D - (An^2 + Bn + Cs + D) = 0$$
$$An^2 + 4An + 4A + Bn + 2B + Cs + Cn + D - An^2 - Bn - Cs - D = 0$$

The coefficients for each term are then collected as follows:

$$(4A + C)n + (4A + 2B) = 0$$

To obtain a set of constraints, the value of each set of coefficients is set to 0. This provides the following constraints from the example verification condition:

$$4A + C = 0$$
$$4A + 2B = 0$$

The verification condition derived from the initial assignments of each variable is also used to generate constraints.

$$T|^{<0,1>}_{<s,n>} = 0$$

$$A(1)^2 + B(1) + C(0) + D = 0$$
$$A + B + D = 0$$

Constraints are accumulated into a system of equations that are solved algebraically:

$$\begin{bmatrix} 4A + C \\ 4A + 2B \\ A + B + D \end{bmatrix} = 0 \rightarrow \begin{matrix} B = -2A \\ C = -4A \\ D = A \end{matrix}$$

Deriving Invariants

Once constraints on the template coefficients have been computed, the are substituted into the original template function.

$$An^2 - 2An - 4As + A = 0$$

To derive a set of invariants, the solver iterates over the remaining free coefficients and sets each to a value of 1 and all other coefficients to 0. For this function, there is only one free coefficient remaining in the reduced template, so the invariant is computed as:

$$(1)n^2 - 2(1)n - 4(1)s + (1) = n^2 - 2n - 4s + 1 = 0$$

Thus, our loop invariant is $n^2 - 2n - 4s + 1$.

Trivial Invariants

If all of the coefficients are reduced to 0, we are left with the trivial invariant $0 = 0$ or *True*. This means that the loop does not have an invariant of the hypothesized form, and the user will be alerted that no invariants could be found.

Accounting for Conditional Statements

In some functions, the condition of an if-else statement can impact the ability to find non-trivial invariants. In the following example, if z is not assigned a value of 0 when the substitutions for the first branch are made to the template, the coefficients are all constrained to 0 and no invariant is found.

Example Conditional Statement

```
if (z == 0){
    y = y-1;
    z = x;
}
else{
    x = x+1;
    z = z-1;
}
```

Currently, the tool is configured to look for if-else conditions of the form $\text{variable} == \text{constant}$, and make the appropriate substitutions to the template functions.

> *Note*: SymPy's solver API has been expanded to allow for the inclusion of some constraints. This will be explored for inclusion in the final version of the tool.

Nested Loops

In the case of nested loops, a separate template function is generated for each loop. The verification conditions for nested loops also mean that the templates that needs to be subtracted following substitutions may change.

Example Function

```
function nexted_example(T){
    m = 0;
    n = 0;
    s = 0;
    x = T;
    while(x!=y){
        x = x- 1;
        n = m+2;
        m = m+1;
        while (m!=0){
            s = s+1;
            m = m-1;
        }
        m = n;
    }
}
```

Verification Conditions

$$\begin{gathered} T_{\text{outer}} = 0 \implies T_{\text{inner}}\big|^{<x,n,m>}_{<x-1,m+2,m+1>} = 0 \\ T_{\text{inner}} = 0 \implies T_{\text{inner}}\big|^{<s,m>}_{<s+1,m-1>} = 0 \\ T_{\text{inner}} = 0 \implies T_{\text{outer}}\big|^{<m>}_{<n>} = 0 \end{gathered}$$

For each verification condition, substitutions are made to the template on the right, then the terms from the template on the left are subtracted from the result. The rest of the process remains the same.

Solving for Invariants that are Inequalities

Solving for invariants using inequality templates (i.e. of the form $T < 0$) has been tested using two methods: as a system of linear inequalities, and using the Fourier-Motzkin elimination algorithm for inequality reduction. While both methods work in certain cases, neither is particularly robust in generating invariants for loops of many different forms.

Support for inequalities will be fixed/expanded in the final version of the tool. This may involve additional work to leverage both methods, using expanding functionality now present in SymPy, and/or exploring the use of an additional/alternative solver such as Z3 for inequality templates.

> *Note*: SymPy's API has been expanded with additional support for the solution of systems of inequalities. This will be explored for inclusion in the final version of the tool.

Tool Outputs

Verification Conditions and Constraints

The tool will output a list of the verification conditions generated by the parser, and all of the constraints that were generated for that VC by the solver.

Example

$T = 0 \implies T\|\|^{<s+n,\ n+2,\ x+1>}_{<s,n,x>} = 0$	$4c_1 + c_3 = 0$ $4c_1 + 2c_2 = 0$ $c_1 + c_2 + c_4 = 0$

Reduced Template Functions

The tool will display the original template function for each loop, and the resulting simplified template after all constraints on coefficients have been applied

Example

Original Template	Reduced Template
$An^2 + Bn + Cs + D = 0$	$An^2 - 2An - 4As + A = 0$

Generated Invariants

Finally, the tool displays the invariants that were generated for each loop.

Example

Loop 1	$\left[n^2 - 2n - 4s + 1 = 0\right]$