SMT Path Feasibility Documentation

Michael Hoffman
Lawrence Livermore National Laboratory

September 10, 2015

1 Developer Info

This is the documentation of the SMTPathFeasibility project source for future developers. We begin with the file analyzePath.cpp and discuss what each SMTPathFeasibility function does as we go.

```
SgProject* proj = frontend(argc,argv);
fixAllPrefixPostfix(proj);
initializeScopeInformation(proj);
SgFunctionDeclaration* mainDecl = SageInterface::findMain(proj);
SgFunctionDefinition* mainDef = mainDecl->get_definition();
StaticCFG::CFG cfg(mainDef);
SgIncidenceDirectedGraph *g = cfg.getGraph();
PathCollector* pathCollector = new PathCollector(g,&cfg);
```

The fixAllPrefixPostfix function replaces unary ++ and – such that the assignment is completed before or after depending on prefix or postfix use
initializeScopeInformation allows ResultQuery naming to avoid problems when we have something like

```
int a = 1;
while (1) {
        int a = 2
        break;
}

a = a + 1;
```

If we don't fix names we will have the a variable assigned three times so that

```
a = a + 1
```

will give 3. If you get an error in compilation always check to make sure initializeScopeInformation was called. The code that follows until PathCollector is standard construction of a CFG for a main function

```
PathCollector* new PathCollector(g,&cfg)
```

The PathCollector is defined in utils/utilHeader.h, it's functions are defined in tools/collectPaths.cpp, we will use it later for analysis.
First we collect all the while statements and see if they have pragma declarations. The while loop

```
SgScopeStatement* scopeOfWhile = SageInterface::getEnclosingScope(*node);
SgStatementPtrList statementsInScope = scopeOfWhile->getStatementList();
SgStatementPtrList::iterator statPtr = statementsInScope.begin();
std::set<SgPragmaDeclaration*> prdecls;
for (; statPtr!=statementsInScope.end();statPtr++) {
    if (isSgPragmaDeclaration(*statPtr)) {
        prdecls.insert(isSgPragmaDeclaration(*statPtr));
    }
}
```

should be straightforward enough.
Now we get to the interesting part

```
SgStatement* body = (isSgWhileStmt(*node)->get_body());
std::vector<std::vector<SgGraphNode*> > paths = pathCollector->getPaths();
std::cout << getPrelude() << std::endl;
SgGraphNode* whileStart = cfg.cfgForBeginning(isSgWhileStmt(*node));
SgGraphNode* whileEnd = cfg.cfgForEnd(isSgWhileStmt(*node));
collectPaths(whileStart,whileEnd, pathCollector);
SgGraphNode* whileOut = getWhileEndNode(isSgWhileStmt(*node),pathCollector);
SgGraphNode* bodyStart = cfg.cfgForBeginning(isSgWhileStmt(*node)->get_body());
SgGraphNode* bodyEnd = cfg.cfgForEnd(isSgWhileStmt(*node)->get_body());
pathCollector->clearPaths();
collectPaths(bodyStart,whileOut,pathCollector);
paths.clear();
paths = pathCollector->getPaths();
std::vector<std::vector<SgGraphNode*> >::iterator i = paths.begin();
std::set<SgVariableSymbol*> vars = getVars(pathCollector);
std::string vardecls;
std::string initrule;
std::vector<std::string> rules = getRules \
(*node,pathCollector, vars, vardecls,initrule);
```

getPrelude is responsible for defined z3 functions that compute c-style integer divisions, mod, conditionals (e.g. 0 = false, 1 = true), and a few more. This returns the string that contains them all, and this is printed to screen so that the output of analyzePath can be piped to a file. In order to make the file readable by z3, just use "head –lines=-3" (due to testing methods the last three lines are unrelated to z3 and should be removed).

```
getVars(pathCollector)
```

getVars generates the set of SgVariableSymbols for any variables occurring in any of the paths.

```
getRules(*node,pathCollector,vars,vardecls,initrule)
```

getRules returns a vector of strings representing the rules which are defined by the paths. At present this only works on while loops showing proof of concept. We send it the node representing the while statement, the PathCollector, all the variables found with getVars, a string to collect all variable declarations, and a string to collect the initial rule.
Next we get to the function getWhileRules loops through all while statements, and calls getWhileRule.

```
std::string getWhileRule(PathCollector* pathCollector,\
 std::vector<SgGraphNode*> path, std::set<SgVariableSymbol*> vars,\
 SgWhileStmt* whileStmt,std::string& vardecls,std::string& initrule) {
        std::map<SgNode*,std::string> eckAssociations = \
        pathCollector->getAssociationsForPath(path);
        SgExprStatement* boundingConditionStatement = \
        isSgExprStatement(whileStmt->get_condition());
        SgExpression* boundingCondition = \
        boundingConditionStatement->get_expression();
        std::string rule;
        evaluatePath(path,eckAssociations,boundingCondition,\
        vars,vardecls,rule,initrule);
        return rule;
}
```

getAssociationsForPath collects any cases of if statements. Consider the if statement

```
if (a == 1) {
        a = a + 1;
}
else {
        a = a - 1;
}
```

There are two paths through this. getAssociationsForPath determines whether a given condition is true or false in the current loop.
next we use evaluatePath, it has arguments path = current path boundingCondition = the bounding condition for the loop (e.g. while (i¡2) returns i ¡ 2 vars = our variables determined earlier vardecls = where our variable declarations will be placed initrule = our initial conditions defined by the user in the pragma evaluate path is defined in getStatementInfoForSMTLib.cpp.

```
        pathStatements.clear();
        for (int i = 0; i < path.size(); i++) {
                if (isSgStatement(path[i]->get_SgNode())) {
                        pathStatements.insert \
                        (isSgStatement(path[i]->get_SgNode()));
                }
        }

        pathValue.clear();
        varExpressionValue.clear();
        variablesAssigned = vars;
        std::set<SgVariableSymbol*>::iterator v = \
        variablesAssigned.begin();
```

```
for (; v != variablesAssigned.end(); v++) {
        getVarName(*v);
}
```

pathStatements here is a global variable that contains the set of SgStatements within the current paths (hence it is cleared before continuing). We get all statements in our given path.
pathValue and varExpressionValue:
pathValue is a map from SgIfStmt* to std::string which associates a given if statement to whether or not the conditional holds in the current path
varExpressionValue is used to carry the current version of the initial variable e.g. variables a, b ...

```
a = a + b;
a = a * b;
```

by the end we have

```
(= a (+ a b)) -> (= a (* b (+ a b)))
b -> b
```

This translates to

```
(rule (=> (R a b) (R (* b (+ a b)) b))
```

That is, given that the loop worked at a, b "(R a b)", then the loop will end with

```
a = (a+b)*b, b = b
```

which is declared via

```
(R (* b (+ a b)) b)
```

Which gives the form of Z3 rules

```
(rule (=> REL_INPUT) REL_OUTPUT)
```

If there is a conditional we append conditions to REL_INPUT, e.g. if $a > 2$ is true in a given path, then

```
(rule (=> (and REL_INPUT (cbool (> a 2) 1)) REL_OUTPUT))
```

cbool here is X Y such that a true condition yields 1, and a false condition yields 0, thus one path would be as above and the path

```
(rule (=> (and REL_INPUT (cbool (> a 2) 0)) REL_OUTPUT))
```

usually this would change REL_OUTPUT, given that generally a conditional would decide if the false body or the true body should be evaluated Example:

```
while(x < 2) {
        if (z > 0) {
                x = x + 1;
        }
        else {
                y = y + 1;
        }
}
```

```
(declare-rel w_rule (Int Int Int))
(declare-var x Int)
(declare-var y Int)
(declare-var z Int)
(rule (=> (w_rule x y z) (< x 2)))
(rule (=> (and (w_rule x y z) (cbool (> z 0) 1)) (w_rule (+ x 1) y z)))
(rule (=> (and (w_rule x y z) (cbool (> z 0) 0)) (w_rule x (+ y 1) z)))
```

This tells z3 that

1 x is always less than 2 at the beginning of the loop

2 if z is greater than zero and the loop condition is fulfilled at x,y,z then x becomes $x + 1$ and y and z do not change.

3 if z is not greater than zero and the loop condition is fulfilled at x,y,z then y becomes $y + 1$ and x and z do not change

2 Pragma syntax

The way the user can give rules to the solver is through pragmas. The syntax for the program are in development, but for proof of concept the prototype has two pragma forms

init pragma smt init [EXPRESSION1,EXPRESSION2,...]

queries pragma smt [EXPRESSION1,EXPRESSION2,...]

Example:

```
#pragma smt init[i=0,a=1]
#pragma smt [a>0,i<10]
while (i < 10) {
```

This means that the loop begins with i = 0 and a = 1. Expressions like $a > 2$ could also be used, implying that the loop begins with $a > 2$, though this will not necessarily hold in successive loops.
The other pragma gives postconditions a¿0 and i¡10, that is that at some point the loop achieves the conditions a¿0 and i¡10.
The pragmas currently in development are similar to the notation used by Spec# (a subset of C#) so we have

assume Only use loop iterations where some condition E holds

assert This is an interior assertion, wherever the pragma is locate

invariant This gives conditions that must hold throughout the loops

requires implemention not yet determined

old Used as $x = old(x) + 1$ in Spec#, not sure how to write this in a pragma

ensures postcondition, pragma implementation is not yet determined

modifies Not sure how to implement

forall Not sure how to implement

3 ANSI C coverage

3.1 Not fully tested or covered

This list is not exhaustive, but describes important points not covered

Unary Ops pre and postfix unary ops

Bit Ops bit operations, bitvectors are not set up for this

Loops do while, for: These should be relatively easy to cover, the evaluation must be slightly modified from the while case

Conditionals ternary if is not yet implemented

Enums Need to check for complete coverage

Case/Switch case/switch is not yet covered

Function Calls

Other structs, pointers, addresses: the problem with these is described later in this document

Typecasting explicit casting is not covered

typedef

3.2 Covered and Tests For Coverage

Basic Arithmetic $+, -, *, /, \%$, integer division is not completely tested

Simple Assignment $=$ is covered

Compound Assignment $+=, -=, *=, /=$

loops while is covered

Conditionals if statements are covered

3.3 Pointers, Structs and Addresses

The code for these will be based on the paper "Efficient Evaluation of Pointer Predicates With Z3 SMT Soliver in SLAM2" T Ball, E Bounimova, V Levin, L De Moura. This requires extensive axioms and definitions. Information can be found in PointerToSMT2.smt2 in docs

3.4 Next Steps

Next steps for this project are as follows...

Directory Organization I have this implemented in an experimental version, but have not yet pushed it to master

DLX pragma format I have this implemented in part in an experimental version, this allows for much greater expression and more complex queries. This uses AstFromString, which seems to have a bug such that && is defined as SgBitAndOp instead of SgAndOp

Extending Pragmas to for loops, do while loops, function call expressions

Basic coverage completion Extending this code to include enums and explicit typecasting, do while and for loops, ternary conditionals, case/switch

Pointers, Arrays, Structs This will require complete implementation of the system referred to in the previous subsection

Tutorials Currently this documentation has a small tutorial, this should be extended to tutorials defined separately.