**RTC Doc**

**RTC and Rose**

RTC is a runtime checking tool for the C programming language that is built on top of the ROSE compiler. ROSE is a source-to-source compiler. It takes source code and converts it to a data structure known as an Abstract Syntax Tree (AST). This tree can then be traversed, analyzed, and edited by using tools written in C++. RTC is one such tool that uses ROSE to analyze C code. Its primary purpose is to keep records of pointers and make checks on pointer arithmetic. This is to avoid the myriad of ambiguous runtime problems one can experience when developing C programs.

**RTC's Implementation**

RTC can be broken down into three parts:

1. Traversal – traverses the tree and adds nodes that should be instrumented to the list located in the “instr” namespace called nodesToInstrument. Relevant files include traverse.C and traverse.h
2. Instrumentation – Iterate through nodesToInstrument and invoke instr on that node. instr is an overloaded function whose various definitions are spread out over several files beginning with instr. The instrumentation can come in the form of metadata (code that tracks the state of a pointer), and checks. main files are instr.C and instr.h
3. Unparse – Unparses the tree into a source code file called “rose\_[nameoffile].C”. (call to unparse takes place in RTC.C where the main function is located)

**Objective: Modify RTC to allow staggered instrumentation**

RTC deploys several checks that impose various levels of overhead. In order to make RTC more viable on low level systems that are frequently part of the Internet of Things, the checks could be staggered (spread out) such that the overhead of only a subset of checks is imposed on the device. Assuming that the devices do send error logs, this could be useful. Even though not every device will have every check, altogether they should have approximately the same coverage as the fully instrumented version. Therefore, the error logs of at least some of the devices should show a bug is present. In order to do this, however, the metadata needed for those checks also has to be preserved. For simplicity's sake, the initial approach was going to leave the metadata untouched and merely focus on taking out the checks.

**Problems**

There is no implementation level distinction between the two types of instrumentation. Metadata instrumentation and check instrumentation are tightly coupled. As is there is no built in way to associate the original nodes and their final instrumentation. One function to examine is “iterateOverNodes” in instr.C. This is where instr is invoked on each node in nodesToInstrument. Once instr is called the node could be passed to several different functions spread through several different files. Any attempt to track the node through these functions will take a great deal of work and has not yet been accomplished. This led to two general approaches by me.

**Approaches**

**1) Use iterateOverNodes**

Use iterateOverNodes and only allow a subset of the nodes to be instrumented. I would use a counter to determine which instrumentation in nodesToInstrument I wanted to use for each traversal of the list. At the end of this traversal, I would unparse and start the traversal of nodesToIterate again with the counter being updated. In order to accomplish this, I still had to traverse the list in it's entirety to ensure that all the metadata instrumentation took place. I did this by using a series of if/else statements that queried the nodes type. If its type indicated that the node was metadata instrumentation (then it would be allowed to instrument).

**Problems**

Unparsing the tree essentially removed the tree from memory. I could not find a means of unparsing that allowed me to unparse multiple times in one run. This is in spite of using various methods from the SageInterface/SageBuilder namespace that were supposed to allow this. Instead I had to make an external shell script that ran the program over and over again, and I had to store the current value for my position in the list in a file. Once this was done, I ran into my next problem.

**Simply swapping a check out isn't feasible at the expression level since the integrity of the tree cannot be guaranteed. This hits at the heart of the problem with staggering the checks. RTC has various fail-safes to help ensure that the resulting tree is valid. When we only allow a subset of the instrumentation to be performed, we circumvent a lot of those failsafes.**

**2) Use a wrapper function for replaceStatement and replaceExpression (current).**

This works on the yet unproven assumption that all edits to existing nodes are done by using these two replace functions from SageInterface. I used a perl script to replace every instance of replaceStatement and replaceExpression with replaceWrapper in the instr files. replaceWrapper is an overloaded function that is defined in instr.C. This is done so that the original code and the final instrumented code can be coupled together.

I created an AST attribute (defined in instr.h) that can be placed on the replacement statement and the original statement. This attribute stores a reference to the replacement (current) and a reference to a copy of the original statement (stored).

When replaceStatement is called I check to see if the original statement already has the attribute. If it doesn't I initialize and place the attribute. If it does, I make a new AstAttribute that uses the original statement from the old attribute and point current at the new replacement statement. I make a new AstAttribute in this case to avoid any confusion about when it is safe to call replaceStatement. You risk taking your statement out of memory if you call replaceStatement too soon.

When replaceExpresion is called I grab the enclosing statement of the original by using SageInterface::getEnclosingStatement(exp) and make a copy (assuming it doesn't already have the attribute). Then I replace the expression, grab the replacements enclosing statement and make a new attribute.

After the instrumentation is done, I have a visitor tool (defined in RTC.C) that visits every node and puts it in a list if it contains my AST attribute. I then go through this list and attempt to “flip” every node except for one to it's original statement. Then I unparse and store my position (as described in the other approach) and start over.

**Current State**

Staggered Instrumentation has thus far been unsuccessful. The current implementation does complete, but all of its output files are identical. Not only that, they are also identical to the non-staggered output file. I have yet to determine where the bug is in my code, and this is where I ran out of time. A copy of my RTC work is in my scratch folder located at iprogress/scratch/scollie. In my scratch folder you will find a folder called “testdata”. This folder contains my test files for staggered instrumentation. A sub-folder called staggeredOutput contains some bash scripts including “autostagger.sh”. Running this with a filename as an argument will run my RTC with the current staggered implementation (NOTE: file paths will probably need to be updated in the autostagger.sh script). The full command currently needed to invoke RTC with instrumentation is

./RTC filename -rose:RTC:instrIndex 1 -rose:RTC:staggered "a"

The “a” is just there to set a variable to a non-null value. This sets the STAGGERED variable to true in the instr namespace. The autostagger script is need for RTC to know where in the list of nodes with the “tempName” AST attribute to keep instrumentation.

**Recommendations**

Simply getting a working prototype that spreads out instrumented statements (excluding metadata) is the first step. This would provide some degree of sample data to benchmark the staggered instrumentation.