IIT Bombay

Design and implementation of GNU Cross Compilation Framework Course Project

QT based plugin for showing specRTL trees pictorially cum manual

Author:
Bharath RadhaKrishnan,
133050014
Sushant Mahajan,
133059007

Supervisor:
Prof. Uday Khedker

April 23, 2014

Abstract

In GCC, the machine descriptions are difficult to read, construct, maintain, and enhance because of the verbosity, repetitiveness, and the amount of details. specRTL[8] provides a compositional specification mechanism for defining patterns that describe RTL templates in machine descriptions. These patterns can be refined by extending them and by associating concrete details with them in a need based manner. Machine descriptions written using specRTL are smaller and simpler and hence easy to read, construct, and maintain. specRTL integrates with conventional machine descriptions seamlessly.

The purpose of the project is to supplement the specRTL description language by providing a pictorial visualization of the patterns which it is capable of describing. The user can browze for an existing specRTL file or create a new one and the tool would produce a n-ary tree annotated with the appropriate attibutes on specific nodes, representing the constructs.

Contents

1 Introduction

Here we describe the problem that we wish to address and the proposed solution.

1.1 Problem Description

specRTL is a language which provides a compositional specification mechanism for defining patterns that describe RTL templates in machine descriptions. The description of the language and its power is not in the scope of the project, but specific explainations will be provided wherever necessary. The specRTL language allows one to create patterns, both abstract and concrete (concrete patterns extend from abstract patterns).
It is however, difficult to visualize the scale and shape of the pattern once it goes beyond a certain size and hence hampers writing complex patterns. A tool is therefore needed to address this issue and provide means of visualizing the tree as it is being written.

1.2 Solution

The goal of the project is to provide a seamless interface where a user can provide the specRTL description and would be presented with the appropriate n-ary tree representing the pattern. We have prepared such a tool using *C++* and *QT graphics library*. It is a GUI desktop applications with the use cases as given below. We have tried to make the application scalable and loosely coupled. The tool supports the following operations:

- Browze for and load external specRTL files.
- Display the tree of the loaded file.
- Edit a previously loaded file and display the tree.
- Create new specRTL files.
- Allow for easy way to browse the patterns in a file, in case it has multiple patterns specified.
- Highlight the pattern being displayed as a tree, in the specRTL source file.
- Should be robust.

- Allows for interactivity with the generated tree by letting the user:
  - Switch the colors of nodes.
  - Allow for zoom in and zoom out.
  - Expand and collapse nodes of the tree.

The parser for the language is not in scope of this project, we have used the parser already written by other students and integrated it with our application. We have however, made changes to the parser in order for it to work in a multi-threaded GUI environment. The details will be provided in the sections below.

2 Software Requirements

Following is the description of the environment in which we developed our application as well as minimum set of libraries which a user requires to run the software.

2.1 Development Environment

- **OS:** Laptop running Ubuntu 12.04.4 LTS(64-bit), Precise Pangolin
- **Editor/IDE:** QT creator (Qt 5.2.1 Open Source) [3]
- **Compiler:** gcc v4.6.3
- **Make:** GNU Make 3.81
- **QMake:** QMake version 3.0, provided with qtinstaller package
- **Parser:** bison (GNU Bison) 2.7 [1]
- **Lexer:** flex 2.5.35 [4]
- **Other libraries:**
  - libqt4-core
  - libqt4-gui
  - libqt4-dev
- **Coffee and Snacks:** Brewberries, Hostel-5 Xerox shop :)

2.2 Release Environment

- **OS:** Any linux based OS, with Qt support, and Desktop Manager
- **Libraries:** (These are specific to Ubuntu (Debian), refer your OS documentation)
  - libqt4-core
  - libqt4-gui
  - libqt4-dev

– If you want to compile the project, you'll also need *make, flex, bison, and gcc* as mentioned in subsection 2.1.

3 Implementation Description

This section provides information on problems faced, the UI, and details of code.

3.1 Problems Faced

- **Parser not invoking from within GUI**

  Initially we tried invoking the parser from the main function without creating the UI process and it worked fine(for a single invocation). Since we require that the parser be called on the event when the user has selected a specRTL source file (of whose tree we are required to show), we found that the parser caused a segmentation fault and the program terminated. It was happening for both large and small specRTL files so we concluded that it was not the source file that was the problem but the way the parser was being invoked as it was working for a single file, when invoked from main without GUI.
  Search on Google did not help and we resorted to hit and trial. We still do not know the exact cause of the parser failing but we thought the since parser worked from outside the GUI environment, maybe on executing it as a separate process or thread we would be able to get it to work. We ran the parser on a thread (*QThread* to be precise), and it started working, though for a single invocation only.

- **Parser not working multiple times, in a single run**

  We are required to parse a specRTL file and generate the tree. The user can select a new file at any time in a run of the program and we have to parse again and display the new tree. But the parser was running only once and on selecting a new file, it was causing a segmentation fault. This was happening for main also. We are invoking the parse method using the same parser object (*srtl_driver* driver*) from within UI.
  This time searching on Google for an answer was fruitful as we came across an old mailing list in which a user was facing a similar sort of an issue[2]. The issue was that on the second run, the *srtl_scanner.cc* was having trouble de-referencing a yylex built-in MACRO. The scanner was not reinitializing so it could not accept a new file as input.
  In order to release the memory after *lexing* of the first file was complete we added a call to *yylex_destroy()*[5], which does precisely the same thing,

i.e. releases memory. We added this call to the *scan_end()* function. After this we were able to achieve multiple runs of the parser.

- **Highlighting the selected pattern**

  Highlighting the displayed pattern in the UI implies knowing the line number where it starts. In its original state, the parser does not report the line number in the line file where pattern is written.
  This information is available with the parser. Without knowing line numbers it is very inefficient to do a search every time and find the pattern name in the file, and highlight it.
  We modified the parser code (yy file) to set the line number identified when parsing, in the *current_pattern* member of the *srtl_driver* class object. This object was already being used in the parser. This also involved adding an accessor and mutator to the *concrete-pattern* class.
  Later we extracted this information and stored it in the object of the class that represents the UI node corresponding to the node constructed by the parser and made the search efficient.

- **Incorrect line numbers reported by parser**

  One of the use cases of this tool were to highlight the text in the source file (visible in the tool), of which the tree is shown (a pattern can be selected from a handy list provided in the UI). In order to achieve this efficiently, we need the line number at which the pattern description starts and highlight it. The line number actually being reported was arbitrarily incorrect. Had it been always ahead or always before, we could have done a search from that point onwards to find the pattern name (which we can get from the list) and perform the highlight.
  The lex code for the parser revealed that in case there were many newlines present in the code, the *yy_lineno* was being incremented only once. Fixing this, gave us a value of the line number that was always a few lines before the correct value.
  Combining this with a string search, gave us the correct pattern location, in an efficient manner.

- **Parser does not provide information on abstract patterns by themselves.**

  Initially, we were displaying only the *concrete_pattern*s. On detection of a standalone *abstract* pattern, the parser was just ignoring. It only considers

the abstract patterns when they are used by other *concrete_pattern*s. We made changes at two places, first it was the *yy* file, to add information to the *currentPattern* pointer in case of abstract patterns, just like we did for *concrete_pattern*s, for getting the line numbers. We also made changes to *abstract.hh* and *abstract.cc*. The *virtual* function *pattern::createPattern()* was overridden so that the parser can call it to fill the structure representing the *abstract* pattern with appropriate information. Because of our flexible back-end, we did not need any change in the structures holding pattern info *util::NodeMap::m_mapVec* and the abstract pattern were included.

- **Miscellaneous**

  We also did some minor modifications like moving the driver code to an more manageable area and creating a pointer to the parser so the entry would be easier. Initially, the parser had its own main and could not be worked with.

3.2 Class Diagram

We have mostly, directly used the UI classes provided by QT directly but made minor changes wherever neccessary.
The new classes made are:

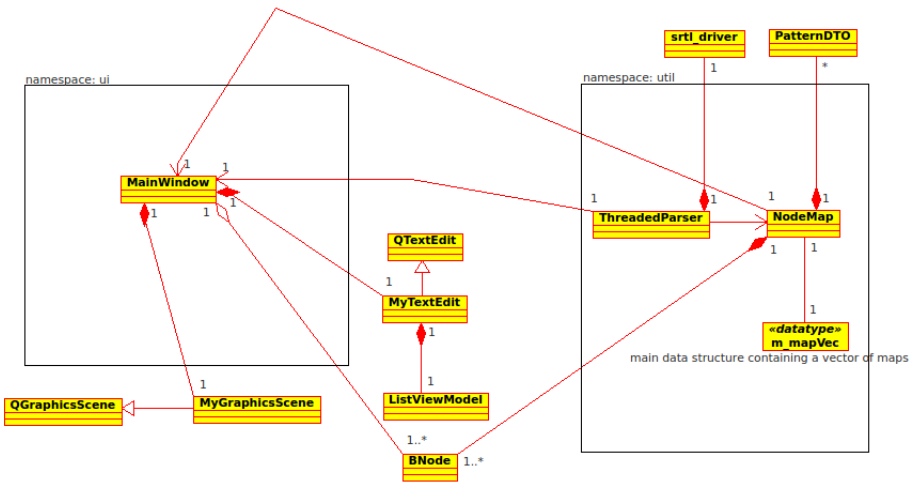
Table 1: New Classes

ui::MainWindow	Class representing the main UI
MyTextEdit	Represents the textarea used for showing and editing specRTL file content, extented from QTextEdit to include keyboard event. [6]
util	Namespace for utility classes and functions.
util::getDriver(int aArgNum, char** aArgs)	Global function, entry point into parser.
util::ThreadedParser	Threaded class to invoke parser multiple times.
util::NodeMap	This constructs a map of nodes displayable on the UI.
BNode	This is a wrapper on the node class, storing extra information related to UI.
ListViewModel	It is the model for the list of patterns shown on the UI for wasy navigation.
PatternDTO	Simple data transfer object class used as a container for Pattern information.
MyGraphicsScene	Derived from QGraphicsScene to support mouse double click event.

Table 2: New header files (interfaces)

mainwindow.h	Class declaration of the main UI class
listviewmodel.h	Declaration for ListView-Model class
bnode.h	Declaration for BNode class
util.h	New namespace util and declaration of getDriver(), NodeMap and Threaded-Parser
listviewmodel.h	Declaration of the model class for MyTextEdit.
pattern_dto.h	Declaration class for Pat-ternDTO explained in above table.
mygraphicsscene.h	Declaration for MyGraphicsS-cene.

Figure 1: High level class diagram of the project



3.3 UI Description

The annotated UI is as follows:

Figure 2: Main UI

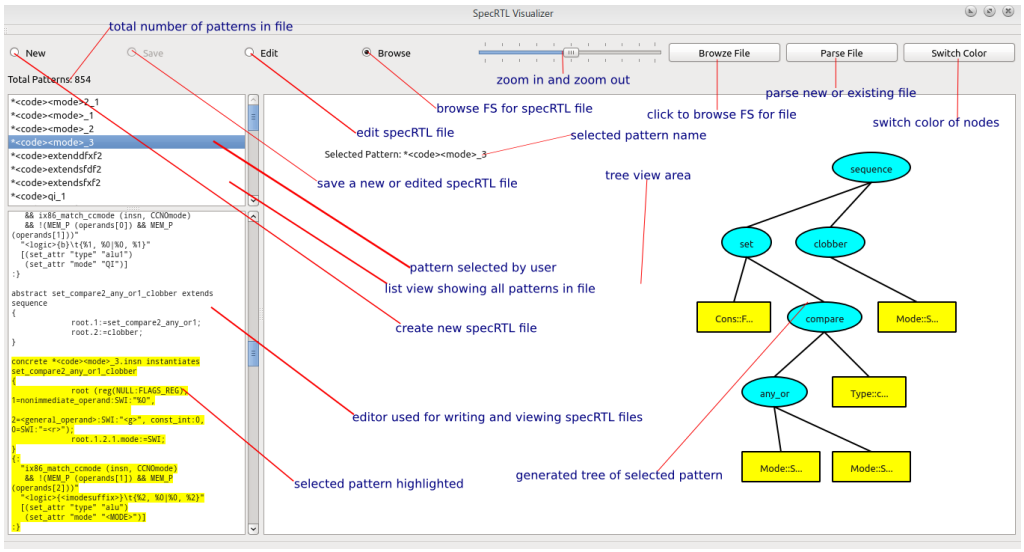


Figure 3: Rendered Tree

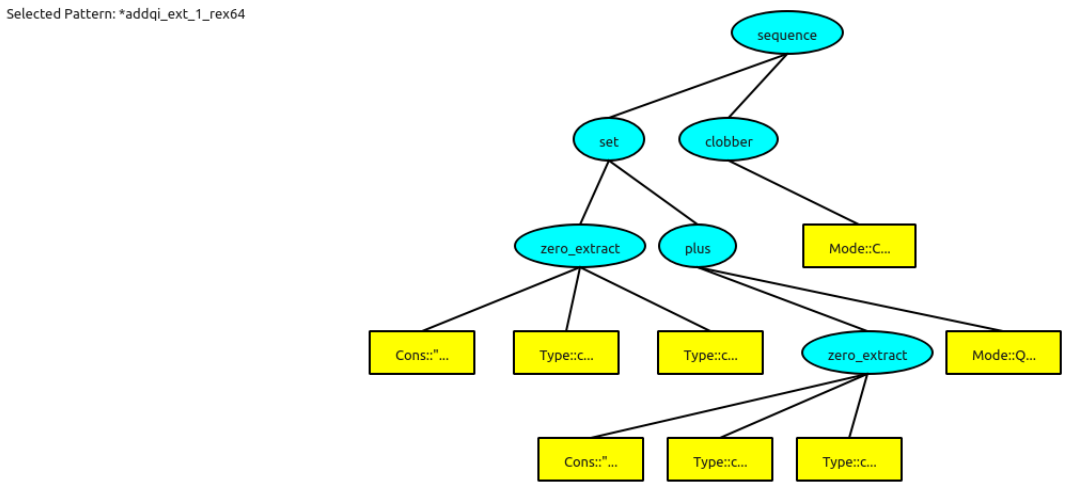
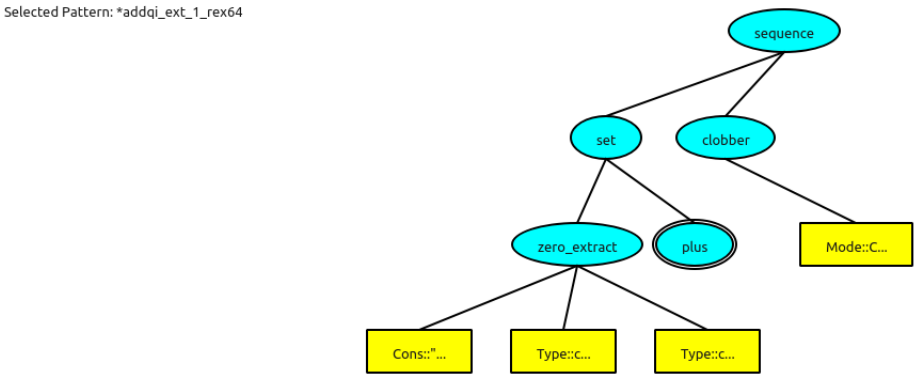


Figure 4: Collapsed Branch



3.4 General Operation

The lifecycle of a new pattern on screen/backend is as follows:

1. The user opens the UI and does either of the following:
   - Clicks the *New* radio button.
     (a) The textarea will become writable.
     (b) The user can input his specRTL description in it.
   - Clicks the *Browse* radio button.
     (a) The *Browse File* push button becomes activated.
     (b) The user can click it and browse for a specRTL file in the filesystem.
2. The user clicks the *Parse File* push button. As a result following actions are taken at the backend:
   - The main UI class *ui::MainWindow* invokes the *util::ThreadedParser* class in a thread and waits for it the parse the file.

- The *util::ThreadedParser* class calls the *util::getDriver(int, char**)* funtion to pass the file name to the backend parser.
- The operation of the parser is not in the scope of this project. It is enough to know that the parser parses the textarea text and generated information in form of objects linked as in a tree and vectors containing additional information. The function returns a pointer to the *srtl_driver* clas.
- Then the *util::ThreadedParser* class calls the *getSymTab()* function on the returned object and it returns a map[7] of all patterns written in the file.
- The operands in the pattern are also fetched by the *util::ThreadedParser*.
- All this information is then set into public static data structures, exposed by the class *util::NodeMap*.
- The map is then iterated over and from each pattern its tree is fetched via *getTree()*, a function which we added.
- The *util::NodeMap* class massages the tree and operands to generate another tree (each node is wrapped in a *BNode*) which can be displayed on the screen.
- *util::NodeMap* first creates a map of vectors for each pattern. The *util::NodeMap::createNodeMap(...)* does this task. The keys of the map are the levels of the tree. In each level we have nodes which are to be displayed at the same level.
- Then function calls a *util::NodeMap::doDepthFirstSearch(...)* on the tree to affix correct operands with each node.
- This map is then added to a vector of maps. This vector of maps is also publically exposed as *util::NodeMap::m_mapVec*.
- The *ui::MainWindow* resumes on completion of these steps and traverses the *util::NodeMap::m_mapVec* to populate the list of patterns.

3. When the user click on a particular pattern in the list, the code using the start line number of the pattern (added in the parser stage), searches the pattern name in the textarea and starts highlighting till the beginning of the next pattern.

4. The tree map for the selected pattern is then iterated over and the information used to draw ellipses and rectangles representing nodes and attributes respectively.

5. On selecting a new pattern the appropriate map is indexed in the *util::NodeMap::m_mapVec* and tree is redrawn.

3.5 Expand and Collapse

This use case means that the user should be able to click (of some other action) on a node in the rendered tree and the branch of the tree rooted at the clicked node should collapse or expand appropriately.
The following allowed us to achieve it:

- We added two *boolean*s to the *BNode* class. These were:

    1. **m_isCollapsed** - to maintain the state whether that node should be visible.
    2. **m_isDoubleClicked** - to remember which node was clicked by the user as that node in contrast to its children in the rendered tree needs to be shown as well as highlighted in some way, so the user can know where he clicked.

- When the user double clicks a node, the event is propagated to the *MyGraphicsScene::mouseDoubleClickEvent(QEvent*)*. From there we can grab the co-ordinates of the click.

- We also store the exacted screen co-ordinates of the items we draw on the screen.

- After the double click, the method *MainWindow::performMutation(QPointF)* is called from the *MyGraphicsScene::mouseDoubleClickEvent(QEvent*)*. *MainWindow::performMutation(QPointF)* then finds the reference to the node at that particular location by comparing the clicked point and the area covered by each node.

- After locating the node, the *MainWindow::mutateBranch(std::map<int,vector<BNode*>>::iterator aM, std::vector<BNode*>::iterator aV, bool aIsCollapsed)* is called. This method sets off the *BNode::m_isCollapsed* flag in all nodes in *MainWindow::m_nodeMap* which are children of the clicked node.

- Then the *MainWindow::createComponents()* is called and it paints the nodes, ignoring those that have *BNode::isCollapsed()* true but not the *BNode::isDoubleClicked*, as that node needs to be drawn as well as highlighted.
- Exactly same thing happens if the node is clicked again to be expanded, except the *aIsCollapsed* passed to *MainWindow::mutateBranch(std::map<int,vector<BNode*>>::iterator aM, std::vector<BNode*>::iterator aV, bool aIsCollapsed)* is false.

4 How to run

Inorder for the program to run for an end user, he needs to have the above mentioned libraries installed. Refer section 2.2
To make the project, do the following:

- Download the project from git hub, `http://github.com/sushantmahajan/gcc\_715` using
  git clone http://github.com/sushantmahajan/gcc_715 whilst inside a local directory.
- Open the terminal and navigate to the root directory, where all the source files are kept.
- On the terminal type, *make clean*
- Then type, *make*
- The output program specviz would be generated, which you can run.

5 Future scope

A lot of improvements can be made to this project, some of which we'll mention here:

- Implement search operation in the text area on the UI.
- Implement search operation in the pattern list on the UI.
- The rendered tree could be more symmetric, perhaps with inclusion of null nodes.
- For the abstract tree it is difficult to check, if it has only one child, whether it is left or right.
- The text editor could be extended to include syntax highlighting(will involve changes to parser).
- Some smooth transitions could be added to the tree when it appears, expands or collapses.

References

[1] *Bison - GNU parser generator*, http://www.gnu.org/software/bison/.

[2] *C++ parser: segmentation fault using parsing*, http://osdir.com/ml/lex.flex.general/2007-01/msg00000.html.

[3] *C++ Qt 81 - QGraphicsView and QGraphicsScene*, http://www.youtube.com/watch?v=b35JF4LqtBs.

[4] *flex: The Fast Lexical Analyzer*, http://flex.sourceforge.net/.

[5] *Init and Destroy Functions - Lexical Analysis With Flex, for Flex 2.5.37*, http://flex.sourceforge.net/manual/Init-and-Destroy-Functions.html.

[6] *Keyboard event and QTextEdit*, http://www.qtcentre.org/threads/25309-Keyboard-event-and-QTextEdit.

[7] *map - C++ Reference*, http://www.cplusplus.com/reference/map/map/.

[8] Ankita Mathur Uday P. Khedker, *specRTL: A Language for GCC Machine Descriptions*, http://www.cse.iitb.ac.in/grc/software/specRTL/sim_gcc_md.pdf, 2011.

Created with LaTeX