Visualization Tools for Understanding and Debugging Maple SDN Programs

Greg Cameron

CPSC 490: Special Project
Advisor: Prof. Y. Richard Yang

May 4, 2017
The Maple system, developed and explained in a recent paper by Voellmy, Wang, Yang, Ford and Hudak, provides a simplified, abstracted and scalable approach to software-defined networking (SDN). While the well-known OpenFlow protocol provides a universal method for routing packets from a programmable switch (which is managed by a separate controller), Maple allows the user to define a simpler routing algorithm than the complex rule-based systems that can arise in large, dynamic networks. This is done through the observation of a single function (titled \( f \) in the relevant paper); the construction of a trace tree to represent routing rules, the optimization of the trace tree to simplify these rules; and the conversion of the tree to OpenFlow rules that can be sent to a given switch.

At its core, programming with Maple is a black box process, meaning the programmer cannot see, and is not meant to see, the inner workings of the system. In Maple’s case, this means the programmer writes the function \( f \) and can observe packets being sent to their correct destinations, but does not see how \( f \) is being converted into a trace tree, and in turn into flow tables. There is a Maple plugin for the Cloud9 IDE — an ongoing project — that lets programmers see some of the details, such as the current state of the trace tree. It also creates all the necessary files and allows programmers to easily deploy their \( f \) functions onto the controller. However, the process between the input (onPacket() function) and output (trace tree or the actual routing of packets) is kept from view of the programmer.

The black box approach has obvious benefits, especially given the complexity of the logic handling and optimization that goes on behind the scenes, but it comes at the cost of more difficult debugging when things go wrong in a Maple application. Because the code being run is significantly more complex than what the programmer sees, standard debugging tools such as

---

breakpoints do not work, and it is difficult for a programmer to see the conversion from the function $f$ to a trace tree clearly.

To help rectify this issue, I develop a framework with which to visualize the relationship between inputs and outputs in the Maple process. The basic flow of a Maple application can be described with this picture:

![Diagram showing the relationship between Code, Trace Tree, and OpenFlow Flow Tables]

The user’s code does not create the trace tree or flow tables itself, but rather, Maple uses the code and incoming packets to “learn” the rules that a programmer has specified in the onPacket() function. The trace tree is thus formed from a combination of code and packets, and the flow tables are formed directly from the trace tree. Because Maple is a combination of all three relationships depicted by arrows, I develop visualizations for each of the three: code-to-trace-tree, packet-to-trace-tree, and trace-tree-to-flow-tables. These three visualizations each help users understand Maple in the context of their own programs, and they also help users debug problems with their code by giving step-by-step information not seen directly in other parts of the Maple plugin.
To provide functionality as similar as possible to a standard debugging tool, these visualizations should give information that is as detailed and flexible as the programmer desires. I provide a few examples of this with features that I have added to the current Cloud9 Maple plugin, and I also suggest additional visualization tools that could be developed.

Code for the two visualizations implemented are located on the Github page for the Cloud9 plugin. The two relevant branches are [www.github.com/snlab/snlab.devopen.server/tree/tracetreehistory](www.github.com/snlab/snlab.devopen.server/tree/tracetreehistory) and [www.github.com/snlab/snlab.devopen.server/tree/codevisualization](www.github.com/snlab/snlab.devopen.server/tree/codevisualization).

**Process**

Initially, the plan for my project was to create an equivalent plugin for the IntelliJ IDE, rather than Cloud9, because of some limitations in Cloud9 that are potentially nonexistent in IntelliJ. However, after speaking with Professor Yang more in February, we decided to plan a visualization framework instead, and we began work on it then.

Because Maple and the Maple plugin for the Cloud9 IDE are part of a collaborative project started long before I began work on it, the first part of my project was a) to understand the Maple system and its purpose, and b) to become familiar with the Javascript, HTML and CSS code in the plugin. Understanding Maple was done with the help of Professor Yang, and building familiarity with the code was done by reading through the code and also consulting Jensen Zhang, one of the current developers for the project, for questions about how best to implement features. I also spent a significant amount of time learning about AngularJS, d3 and jQuery, all of which were new to me but are crucial in the data flow and rendering of the trace tree.

After all the preparation and planning was complete, I then began an iterative process that involved discussing a framework with Professor Yang, adding code to the plugin, and consulting
with Dennis Tao for help with server-side adjustments to the Maple controller. Because I was
given access only to the client-side code and because I had a 12-hour time difference with Jensen
and Dennis, fast communication was often hindered, and it was difficult to complete all the
server-side changes that were necessary to get the correct information to the plugin on the client
side. Ultimately, however, I did get most of the changes that I needed to make progress.

Examples of Visualizations

Visualization 1: Packet to trace tree

The first visualization is a simple view of the current trace tree’s history, as viewed
through the sequence of packets sent through a controller. Users can step through the history of
packets sent since the deployment of their program and view the state of the trace tree just after
any packet was sent. Because trace trees are built path-by-path, with each packet adding either
zero or one “paths” through the tree, observing each packet that is sent through the controller can
help modularize the process to the programmer. Each packet can thus be seen as a “breakpoint,”
akin to the same feature in most debugging tools. A user can identify the first packet that causes
the undesired trace tree and explore this part of the trace tree and program accordingly. In order
to visualize the change made in the trace tree even more clearly, the path corresponding to each
packet is highlighted in the history view.

To implement this, I added a “Trace Tree History” tab to the controller view of the Maple
plugin, and had a trace tree history API call added to the Maple controller server. The new API
takes as input a step number $n$ and returns the state of the tree just after the $n$th packet was seen
by the controller, as well as information about the packet. The beginning of an example response
looks like this:
The new “Trace Tree History” tab looks similar to the “Trace Tree” tab, which shows the tree in real time, except with an added header to show the packet whose corresponding trace tree the user is viewing, and a timeline giving an overall view of all packets. Code for rendering the tree is shared between the “Trace Tree” and “Trace Tree History” tabs, to avoid code duplication. However, in order to make the tree re-render successfully when the sequence number is changed, I first needed to fix a bug in which tree updates were simply stacking on top of old trees. This bug was previously adding a tree every second, meaning that if a tree view was kept on for a long time the page could contain thousands of identical trees stacked on top of each other. I also fixed a bug in the drawTree() function that kept lines from appearing between nodes.
Specific to the history rendering, the new feature requests a new tree from the server every time a sequence number is selected by either specifying a packet directly or pressing the “Next” or “Previous” links. The JavaScript code running in the background caches the sequence number and the tree being shown, so that it is easy to tell if a new tree is actually the same as the tree already being shown. To highlight the path of the current packet through the tree, I used the node and link data and followed the logic of the tree to find the next node recursively at each step. That is, starting at the root node, first the code finds the attribute a node looks at, then it finds the correct link for the current packet, follows the link to a new node and repeats, until a terminal node is reached. All information necessary for this process is contained in the main Maple API call. The packet information, including all necessary attributes, is located in JSON form in the “pkt” field, the trace tree nodes are found in “tt-node-v2,” and the trace tree links are found in “tt-link-v2.” Finally, the d3 “fill” function is used to color all nodes along the path yellow, like so:
Possible enhancements to this feature are a header with more information, especially when getting a view of all packets. The Vis.js timeline library was the best option I could find to show all packets at once in a clickable format, but the tiny differences between packet timestamps make for awkward overlap of points in all timelines libraries I saw. Other additions could include better animation using the d3 library when rendering trees. For instance, when a user steps forward in time and a node gets added to the tree, this could be shown as two nodes “splitting” from the old node, rather than a re-rendering of the tree.

Visualization 2: User code to trace tree

If a user identifies an undesired change in the trace tree and the packet that caused it, he or she may still understand the part of the original code that produced the error, particularly if the onPacket() function is very complex. The second visualization tries to solve this issue by showing, for each node in the history view, the line of code and line number in the onPacket() function (or in a function called from the onPacket() function) that made each decision. For instance, if a node compares the source IP address to 10.0.0.1, a mouseover on that node will show the Java code that did the same — comparing the source IP address to 10.0.0.1. In complex programs, where attributes can be looked at in several instances in a dynamic way, this can help users pinpoint a specific part of their program that is causing a problem.

To implement this feature, a “debugInfo” item was added to each node of the trace tree returned by Maple, like so:

```json
"debugInfo": {
  "filename": "MapleApp.java",
  "method": "onPacket",
  "classname": "org.opendaylight.mapleapp.impl.MapleApp",
  "linenumber": 54,
  "callMapleAPI": "IPV4SrcCls"
}
```
Given this information, I added a mouseover overlay to each node in the trace tree history view, so that users can see, for each node, what dependency Maple recorded in order to infer the decision at that node. When a node at which a decision is made is hovered over in the trace tree history view, the window looks as follows:

It will aid in explanation to examine the source of the “debugInfo” information in more details. The line specified at each node is the line that caused Maple to record a dependency for a decision at a node. For instance, if the following block of code fires for a given packet:

```java
24   if (pkt.IPv4SrcIs("10.0.0.1")) {
25       pkt.setRoute(path);
26   }
```

the plugin would identify line 24 of this file as the line that set a dependency, and the user can examine this line in context to see why a decision is being made as it is.

This doesn’t necessarily tell the whole story, however, as dependencies can exist in ways that are more complex than this simple method. Consider the following example:
In this case, the plugin would note line 10 as the most important line in this decision, but really lines 10-12 are all crucial. As programs get more complex and have more classes and/or functions called, finding the dependencies naturally gets harder for the plugin to do automatically. There must be an extent to which it is up to the programmer to look through the code and sort out the most relevant lines. Still, we would like some way to visualize these code dependencies to make it easier for a programmer to find the source of bugs. Here I suggest one possible method to use, which could be expanded upon for future work on a plugin such as this one.

A simple, brute force-like way to track dependency on a variable (such as `ethType` above) would be to assume that every time a relevant variable is used during the assignment of a new variable, the new variable is dependent on the relevant variable and is thus relevant. An implementation of this would look like the following pseudocode, to be called on lines of code after the first dependency is tracked:

```
function findImportantLines(initialVariable, remainingCode):
    importantLines = {}
    dependentVariables = {initialVariable}
    for line in remainingCode:
        if line contains 1 of relevantVariables:
            importantLines.add(line)
        if variable assigned in line:
            dependentVariables.add(assignedVariable)
    return importantLines
```
This method captures dependencies naively, but it would work in identifying that, for instance, lines 10-12 are relevant in the example above. The important lines could then be highlighted in the Cloud9 IDE (or in any IDE, as this is not specific to a Cloud9 plugin) for the user to identify which parts of the code are important for a specific decision. Importantly, this pseudocode cannot handle function calls within code. If a relevant variable is passed as a parameter to a function, the script will not go inside that function to find more important lines. This could be a limitation, as there may be important lines within a function that is called by the outer code, but it also could be a benefit. Users would be wasting time if they were shown all the lines in
\texttt{pkt.setRoute(path)}, for example, or \texttt{Integer.toString().}\ The conservative tactic presented here, while limiting in its inference, has the benefit about making no assumptions about the code and relying on the user to provide some level of human intuition. Other methodologies for this feature are, of course, possible.

A conservative implementation of this method is provided in JavaScript under the file name \texttt{codeDependencyTracker.js}. It was tested on several simple inputs and does indeed pick out the correct lines.

\textit{Visualization 3: Trace tree to flow table}

A key idea of Maple is that the trace tree, which functions as a decision tree, efficiently captures all the information that the controller knows about the user-specified flow rules. Because switches use flow tables in OpenFlow, however, the trace tree needs to be converted into a flow table at each switch so that switches can begin making efficient routing decisions.

For users who are familiar with OpenFlow but new to Maple, it may be useful to see how Maple is completing this final step. Seeing the live flow tables in text format at any switch is
currently possible using mininet, but a more detailed visualization would provide a clearer and more informative picture. For instance, how does the complexity of the flow tables change over time? Or how does the complexity of the flow tables grow with the complexity of the trace tree? These questions and others cannot be answered easily with live flow tables shown in a terminal window, but they could in a more flexible IDE environment.

Fortunately, building an API that sends flow tables to the Cloud9 plugin would be straightforward. In the Maple paper, Voellmy et al. concisely shows the way a trace tree can be converted into a flow table “by recursively traversing the [trace tree].” The pseudocode for this is as follows:

```
function buildFlowTable(t):
    priority = 0
    buildFlowTableRecursive(t, any)
    return

function buildFlowTableRecursive(t, m):
    if type(t) == L:
        emitRule(priority, m, value(t))
        priority = priority + 1
    else if type(t) == V:
        for v in keys(subtrees(t)):
            buildFlowTableRecursive(subtrees(t)[v], m & {attr(t) : v})
    else if type(t) == T:
        build(t_, m)
        m_t = m & {attr(t) : value(t)}
        emitRule(priority, m_t, ToController)
        priority = priority + 1
        build(t_+, m_t)
```

---

A few explanatory notes for this pseudocode are necessary:

1. Flow table rules are tested in decreasing order of priority. Because this algorithm guarantees that we increase the priority every time we create a rule, the last rule created has the highest priority, the first has the lowest, and we get unique priorities for every rule.

2. The parameter $m$ describes all the match rules that need to be tested at the current point in the tree. This way, when we reach an end node of the tree, of type $L$, we can use $m$ to create a flow rule with all the necessary matching attributes up to that point. $m$ starts at “any,” which essentially means $m$ is empty at the top of the tree and does not require any matches.

3. The $\text{type}(t) == T$ case is somewhat complex due to limitations of flow tables. Because flow tables do not support matching on negations of attributes, such as $\text{pkt.IPv4Src} \neq 10.0.0.1$, a special trick is needed. First the rules for the negation subtree are built, with the lowest priority in this part of the tree. Then the $\text{emitRule}(\text{priority}, m_t, \text{ToController})$ creates a “barrier rule” with one higher priority, which guarantees that if any packet matches all attributes up to the current node and also tests positive for the attribute at the current node, it will never get to the part of the flow table that applies to the negation, because at the very least the packet will be sent back to the controller. Finally, the more specific rules for the positive test are added with even higher priority.

Because this logic is already being handled on the Maple controller, it would make the most sense to create an API call, similar to the trace tree history, on the server side rather than on the client. In this way, a) the code is not reliant on the specifics of the trace tree API, b) an optimization in the conversion from trace tree to flow tables will not require an update to the plugin, and c) the feature could then be easily added to other plugins, such as a potential IntelliJ
IDE. Though such a feature could not be added to the Maple controller in time because I was not able to be granted access to the Maple code, here I discuss details of the client-side feature, if the API were available.

The controller view of Cloud9 plugin could have a new tab, just like the “Trace Tree History” tab that I created, to view flow tables historically (by packet) alongside the corresponding trace tree at each stage. By viewing the trace tree alongside flow tables, the user can visualize the relationship between the tree and the tables and the way that Maple is creating the flow tables for the final step in its process. In this case, however, there are two axes of variation, not just the lone packet axis that the trace tree history view had. Flow tables are installed at each switch, rather than the overall trace tree, which shows the full paths (switch-to-switch) for all packet combinations. Thus we want to be able to view multiple flow tables for each step in the packet timeline, rather than just one.

The packet selection tool could be the same as that used in the trace tree history tool, with “Next” and “Previous” buttons and/or a dropdown to select from the list of packets that have been sent. Below it, I propose having two equally-sized boxes that are side-by-side horizontally, with the box on the right being a (cropped) version of the d3 window shown in the trace tree history tab, and the box on the left having one or more corresponding flow tables, in an easy-to-read HTML table format. There are two options from here:

1. Show one flow table at a time, with tabs to change the switch for which the user is viewing flow tables. In this case, the trace tree could actually be changed to also show the decisions made for each switch. That is, while the overall trace tree has its terminal nodes show a path, the switch-specific trace tree could have its terminal nodes show the next destination in the path, such as “openFlow4:1”. This implementation would look like the following:
2. Show all flow tables at once, with the overall trace tree in the right box. The left box could be scrollable to fit all flow tables as necessary. This implementation would look as such:
The first implementation would likely be the most informative, offering details that are more orthogonal to the trace tree history view, such as the switch-specific trace tree. The second implementation, however, would be simpler and perhaps easier to understand. An ideal solution would potentially contain both options, with the “all switches” option being a tab alongside the tab for each individual switch.

**Future Research**

Due to logistic limitations, I was not able to directly modify the Maple API, which sends information about the controller, such as the trace tree, to the Cloud9 plugin. Instead I communicated with Dennis Tao to make the necessary changes, such as the trace tree history API. If more work could be done on the Maple controller API, plenty of additional features could be added to these visualizations. For instance, the codeVisualization.js script, or a script based on a different methodology, could be applied on the server to show more intricate code dependencies than those shown in the feature I made, or the entire line of relevant code could be sent to the client from the controller, rather than simply the line number. This last feature is something I planned on adding, but Dennis did not have a chance to make the API change in time. Another potential feature is an implementation of the third visualization. The buildFT() function could be used on the server to make the third visualization as a new tab on the client-side plugin.

In addition to the visualizations that I laid out here, other useful tools could help users in debugging and understanding their Maple programs. For instance, a feature could be added that allows users to make small modifications to their code and preview the trace tree for the new code. Because the Maple controller already stores the packet history, this is possible with time:
Just run through the packet history in the same order under the new deployment, send the new trace tree to the client, and discard. This would help users try out different versions of their code without resetting the controller and sending all new packets, which is the necessary process currently.

Acknowledgements

I would like to thank my advisor, Professor Yang, for suggesting the project in the first place and meeting with me several times throughout the semester to discuss overall ideas for the project. I also am grateful to Jensen Zhang, who helped me tremendously in understanding the code of the Cloud9 plugin, and Dennis Tao, who was very responsive and helpful in developing the server-side changes for these features, despite being 12 hours ahead of me in Shanghai.