Automated Visualization for Flat and Hierarchical State Machines

Jasmine Lesner, Gabriel Hugh Elkaim

***Abstract*—Finite State Machines (FSMs) are crucial for event-driven control systems, enabling simplified decision-making through state transitions. However, the increasing complexity of FSMs, marked by the addition of states and events, significantly complicates debugging and feature integration. Traditional state diagram tools require manual inputs or source code annotations, making them susceptible to errors and inefficiencies. This paper introduces an innovative tool that automates the generation of accurate state diagrams from FSM source code. The tool leverages naming conventions and Abstract Syntax Tree (AST) patterns, utilizing a pipeline of XSLT transformations. It offers full automation for standard coding practices, while providing flexibility for non-standard conventions through customizable XSLT templates. This approach allows users to adapt the tool for different coding styles and enhances the process of designing, debugging, and updating FSMs, ensuring that the visual representations always align with the implemented code.**

***Index Terms*—Finite State Machines (FSMs), Automated Visualization, State Diagram Generation, Source Code Analysis, Abstract Syntax Tree (AST), XSLT Transformations, Event-Driven Control Systems, Debugging and Feature Integration, Coding Conventions, Software Tools for FSMs.**

I. Introduction

FINITE state machines (FSMs) are ideal for reactive event driven control because they reduce decision-making to handling events and switching states however with every extra state and every extra event the number interactions can grow exponentially so designers of FSMs can find it challenging to trace bugs in their FSM implementation and to add new features not originally anticipated.

A. State Diagrams

A state diagram can provide a high level map of how an FSM operates. Having a map to navigate FSM logic helps developers design, debug and update FSMs however creating and updating such a map manually is a tedious task and due to human errors and feature creep one can never be sure a state diagram matches the code that implements the FSM.

B. Diagram Tools

Visualization tools like Graphviz [1] (also Mermaid.JS [2], PlantUML [3], ...) require diagrams to be already described using their visualization language. Tools like Doxygen [4] require source code to be annotated for state diagram generation. Unified Modeling Language [5] IDE tools like "Enterprise Architect" [6] need manual intervention for FSM diagram creation. No tool found can automatically generate diagrams directly from source code.

Autonomous Systems Lab, University of California, Santa Cruz, CA 95064
https://asl.soe.ucsc.edu/home

C. Automatic Diagrams

FSM code typically involves a series of checks: current state, last event, event parameters, and guard conditions. Implementations can vary, using structures like switch-default or if-elseif-else statements, and the sequence of checks can differ. This variability poses a challenge: **How can we automatically generate accurate visual representations of FSMs from their source code?**

To address this, we developed a tool that extracts state diagrams from source code. It uses naming conventions and Abstract Syntax Tree (AST) patterns, employing a pipeline of XSLT [7]. This tool is fully automated when standard code conventions are followed. For non-standard conventions, it offers flexibility through modifiable XSLT templates. Users can adapt the tool to alternative naming conventions either by altering the XSLT directly or by preprocessing the source code. When encountering unfamiliar variable names and coding styles, the tool's AST pattern recognition can be expanded with new or updated XSLT templates. This approach ensures that any enhancements in the diagram generation process are immediately reflected across all diagrams, facilitating efficient and accurate visualization of FSM implementations.

II. Method

The tool operates in three stages:

1) The first stage reads source code and generates an abstract syntax tree AST
2) The second stage analyzes and annotates the AST with tags relevant for state diagram.
3) The third stage uses the AST tags to generate a diagram description which is then rendered visually in various formats (PNG, SVG, PDF)

A. Stage One: AST Generation

1) Supported Inputs: We designed our tool to interpret FSMs in an embedded C variant for PIC32MX microcontrollers, a cost-effective 32-bit MCU family with versatile memory and integrated peripherals. This technology is used in UCSC classrooms for developing robotic applications with Microchip's MPLAB X IDE [8] and MPLAB XC Compilers [9], ranging from basic movement to complex autonomous functions.

2) Keywords and Constructs: The embedded C variant for PIC32MX microcontrollers uses C language elements like `va_list`, `__attribute__`, and `__extension__`, which are not recognized by some parsers like PycParser [10]. These elements, unnecessary for our diagram generation, are eliminated

```
find "$src_path" -type f -name '*.c' -print0 \
    | xargs -0 egrep -l nextState \
    | while read f ; do
       ff="`basename \"${f}\"`"
       b="`dirname \"${f}\"`"
       (
           cd "$b" \
           && echo "amalgamating '${f}'" \
           && cat "${ff}" \
                | dos2unix \
                | perl -p "${epath}" \
                | ( egrep -avi '^#define '  || true ) \
                > "${ff}.undef" \
           && echo "
                cpp
                -I\"${course_include_path}\"
                -I\"${pic32mx_include_path}\" \
                $ilist $iconfig2 -I'${b}' -I. '${ff}.undef' \
                " \
                | bash \
                | perl -pe '
                    s{zz0912819zz}{}g;
                ' \
                | dos2unix \
                > "${ff}.cp5" \
           && rm -f "${ff}.undef"
       ) 2>&1
    done

find "$src_path" -name '*.c.cp5' \
| while read f ; do
echo "visualizing '$f'"
(
    sx=saxonb-xslt
    cat "$f" \
        | tr -d '\r' \
        | ( egrep -avi '^[[:blank:]]*$|^#|va_list|__attribute__' || true ) \
        | perl -pe's{__extension__}{ }g; s{__}{}g; ' \
        | python3 c_ast_xml.py \
        | tee "${f}.xml" \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00005_identity.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00100_declutter_attributes.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00200_add_bLine_eLine.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00300_add_CurrentStateTest.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00300_add_EventParamTest.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00300_add_EventTypeTest.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00300_add_NextStateLabel.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00400_add_CascadeElements.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00500_add_CascadeLabel.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00550_add_EventLabel.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00560_add_Guard_Element.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00570_add_Guard_Attributes.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00600_add_onEntry_onExit.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00600_add_onTransition2.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00620_drop_unwanted_code.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00800_gv_digraph4.xml \
        | perl -pe 's/ && / &amp;&amp; /g;
                    s/ < / &lt; /g;
                    s/ > / &gt; /g;
                    s/ <= / &lt;= /g;
                    s/ >= / &gt;= /g;
                ' \
        > "${f}.gv"

        dot -Tpng "${f}.gv" -o "${f}.png"
        dot -Tpdf "${f}.gv" -o "${f}.pdf"
        dot -Tsvg "${f}.gv" -o "${f}.svg"
) 2>&1
done
```

Fig. 1. Two principal commands power our tool. The first command (spanning lines 1-28) prepares C code for parsing. During preparation some macros are protected from expansion (line 11) and after cpp this protection is removed (lines 21-23). The second command (spanning lines 30-69) builds the AST (line 39), runs the annotation pipeline (lines 41-55) and generates diagrams (lines 56-67).

using regular expressions. Additionally, superfluous elements such as empty lines and comments are also removed.

3) Macro Encoding: C programs use macros, e.g., `#include "stdio.h"` and `#define FRONT_BUMBER 0x42`. These are processed by the C Preprocessor (CPP) [11], which enables macro functions, file inclusion, and conditional compilation. The `#include` macros need merging, and `#define` macros replace text in the code. In diagrams, it's beneficial to display macro names like `FRONT_BUMBER` instead of their expanded forms (e.g., `0x42`). Therefore, our prototype selectively suppresses some macro expansions during CPP processing. This is achieved by protecting them from expansion, and later removing this protection. The protection is done by an adhoc script the generation of which is shown in figure 2.

```
epath="$src_path"/encode.pl

find \
    "${src_path}" \
    "${course_include_path}" \
    -type f \( -name '*.h' -o -name '*.hpp' -o -name '*.c' \) \
    | tr "\n" "\0" \
    | xargs -0 cat \
    | dos2unix \
    | ( egrep -ai '^#define' || true ) \
    | perl -pe 's/#define (\w)(\w+)[ \(].*$/s{\\b$1$2\\b}{$1zz0912819zz$2}g; #
     encode123 /g;' \
    | ( grep encode123 || true ) \
    | perl -pe 's/ # encode123//g;' \
    | sort | uniq \
    > "${epath}"
```

Fig. 2. Generation of adhoc encoding script to protect macros

4) Apply CPP: After filtering out unsupported keywords and encoding macros, we use CPP to expand #include files. Post-CPP, the macro protections are removed, reverting them to their original names.

5) Construct AST: A Python script processes the CPP output, creating an XML with two sections: "code" and "ast". The "code" section lists the input C code with line numbers, useful for diagram annotations. The "ast" section contains the corresponding AST, as generated by PycParser.

B. Stage Two: AST Annotation

In this stage, we annotate the AST using a series of XSLT steps, facilitating independent inspection and development of each annotation phase.

1) XML Normalize: Initially, we normalize the XML AST to enhance readability and track changes more efficiently. This involves removing unnecessary whitespace and maintaining the integrity of all XML elements and attributes. Indentation is used for clear visualization of the AST's tree structure.

2) AST Declutter: We simplify the AST by removing redundant elements and attributes generated by PycParser that are not required for state diagrams. Attributes like `quals`, `align`, `storage`, `funcspec`, and `line` (when null) are omitted, along with any empty attributes, using targeted XSLT rules. This decluttering focuses on creating a cleaner, more navigable AST.

3) bLine / eLine: Each AST element is assigned `bLine` and `eLine` attributes, marking the start and end line numbers in the original C code, respectively. This facilitates linking AST elements to their corresponding source code lines, essential for illustrating logic in state diagrams.

4) CurrentStateTest: For `case` and `default` elements within `switch` statements checking `CurrentState`, we add a `CurrentStateTest` attribute, reflecting the state name represented by that case. This annotation is extendable to `if-elseif-else` patterns if encountered.

5) EventParamTest: We tag AST elements within conditional statements involving `EventParam` with an `EventParamTest` attribute, indicating the specific `EventParam` being tested.

6) EventTypeTest: Similar to `EventParamTest`, conditional statements involving `EventType` are tagged with an `EventTypeTest` attribute, specifying the `EventType` under consideration.

7) NextStateLabel: Elements indicating state changes (class `Assignment`, operation `=`, and `nextState` on the left side) receive a `NextStateLabel` attribute, denoting the new state as defined in the assignment's right-hand value.

8) CascadeElements: `Case` and `Default` elements following uninterrupted `Case` elements (without a `Break`) gain `CascadeElement` children, representing each cascading case value.

9) CascadeLabel: A `CascadeLabel` attribute is formed by merging the current case value with all `CascadeElement` values, separated by `" or "`. This label collectively represents switch branches that cascade together.

10) EventLabel: Elements with `NextStateLabel` are also tagged with an `EventLabel`, combining relevant `EventType` and `EventParam` values.

11) GuardElements: `If` statements leading to state transitions but not checking Event attributes are marked with a `guard` child element, encapsulating the condition's code. This highlights the triggering logic in diagrams.

12) GuardLabel: To uniquely identify guards, we use `CurrentStateTest` and `NextStateLabel` attributes, with the guard's line number serving as an identifier. The `EventLabel` differentiates true and false conditions.

13) onEntry / onExit: `onEntry` and `onExit` elements are added, populated with code executed upon entering and exiting states, respectively.

14) onTransition: The `onTransition` element, filled with code executed during state transitions, is added. This information is displayed alongside event labels in the state diagram.

15) Code Declutter: We remove code lines that are redundant or non-essential, such as references to `nextState`, `makeTransition`, and `ThisEvent.EventType`. This is because their actions are already represented diagrammatically.

C. Stage Three: Diagram Generation

Once AST annotations are applied they are used to generate a description of a diagram in GraphViz diagram description language. This is done in four steps by XSLT in figure 3:

1) Step: Diagram Setup: Output format is set to plain text, suitable for Graphviz format and the initial starting state for the diagram is identified.

2) Step: Loop over States: We loop through AST elements representing different states, excluding the initial state and guard conditions. These are formatted with matching styles and labels including `onEntry` and `onExit` code blocks.

3) Step: Loop over Guards: We loop through guard conditions associated with state transitions, adding them to the digraph with their specific style.

4) Step: Loop over Transitions: Last we loop through state transitions adding them to the diagram description with their `onTransition` code blocks.

III. Results

A. Input & Output Samples

Figure 6 displays the state diagram generated from FSM code in figure 5. This FSM, representing the primary level in a hierarchical state machine (HSM), controls a wheeled robot modeled after a cockroach (shown in figure 4). It exhibits behaviors like moving in darkness and freezing in light, with an added periodic 'jig dance'. While each top-level HSM state contains a nested FSM, these are omitted for brevity.

Figure 8 presents the FSM derived from the code in figure 7, which is a lower-level FSM in a multi-tiered HSM for a competition robot. This complex FSM includes labels like `if(barrierCount < BARRIER_COUNT)` and `if ((fieldSide == FIELD_LEFT)...` demonstrating our prototype's capability to manage even chained state transition guard conditions. The diagram also exemplifies the labeling of state diagram elements with corresponding source code.

B. Prototype Benchmarks

The prototype underwent benchmarking on WSL2 Ubuntu Linux on top of a Windows 10 Pro host, powered by an Intel Core i7-8850H CPU. This setup features six physical cores, with twelve hyper-threaded virtual cores, operating between 800MHz and 4200MHz.

Table I lists the outcomes of four benchmark runs, each time processing identical code files to generate thirteen state diagrams. Two of these diagrams are shown in figures 6 and 8 generated from code in figures 5 and 7. The tests were conducted on a laptop plugged into AC power, using Windows 10 Pro default power profile settings. During the tests, three virtual cores were occupied with background tasks, leaving nine cores primarily for our prototype.

The benchmark results indicate that:

1) **Average Diagram Generation Time:** It takes approximately 7 seconds to generate one state diagram, with a 20-25% time variation between the fastest and slowest runs. This discrepancy is likely due to thermal throttling affecting CPU performance.
2) **CPU Utilization vs. Elapsed Time:** Contrary to expectations, higher CPU utilization did not correlate with shorter elapsed times. The longest processing time coincided with the highest CPU usage, suggesting that thermal throttling might have slowed down the cores, increasing the overall time despite higher resource usage.
3) **Core Utilization Efficiency:** The prototype utilizes eight of the nine available virtual cores, leaving limited scope for further parallelization on our test system. While servers with more cores might benefit from concurrent diagram generation, our users (UCSC students) are unlikely to see significant performance improvements on standard laptops or PCs from additional parallel processing capabilities.

TABLE I
BENCHMARK TIMING RESULTS

Run	Percent of CPU	Elapsed Time
Run 1	821%	1:21.22
Run 2	808%	1:17.26
Run 3	819%	1:35.38
Run 4	816%	1:16.48

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text"/>
    <xsl:template match="/">
        <xsl:variable name="InitState" select="//ext[@class='Decl' and @name='
      CurrentState']/init/@name"/>
        <xsl:text>
digraph fsm {
    </xsl:text>
    <xsl:value-of select="$InitState"/>
    <xsl:text>[shape = "point", color = "black",style="filled",width=.1,forcelabels
      =false];

    // states
    node [shape=plaintext]
    </xsl:text>
        <xsl:for-each select="
            //*[
                @CurrentStateTest
                    and not(@CurrentStateTest = '')
                    and not(@CurrentStateTest = $InitState)
                    and not(guard)
                ]">
            <xsl:text>
    </xsl:text>
            <xsl:value-of select="@CurrentStateTest"/>
<xsl:text><![CDATA[ [label=<<TABLE BORDER="1" CELLBORDER="0" CELLSPACING="0" style
      ="rounded">
        <TR>
            <TD BORDER="1" SIDES="B">]]></xsl:text>
            <xsl:value-of select="@CurrentStateTest"/>
            <xsl:text><![CDATA[</TD>
        </TR>]]></xsl:text>
        <xsl:if test="stmts[@class='Assignment' and rvalue/args/exprs/@name='
      ThisEvent']/lvalue[@name='ThisEvent']">
            <xsl:text><![CDATA[
        <TR>
            <TD ALIGN="LEFT">]]></xsl:text>
        <xsl:value-of select="stmts[@class='Assignment']/rvalue/name/@name"/>
        <xsl:text><![CDATA[</TD>
        </TR>
            ]]></xsl:text>
        </xsl:if>
        <xsl:text><![CDATA[
        <TR>
            <TD ALIGN="LEFT"><B>/Entry: </B></TD>
        </TR>]]></xsl:text>
            <xsl:for-each select="onEntry/line">
                <xsl:text><![CDATA[
        <TR><TD ALIGN="LEFT">]]></xsl:text>
                <!-- <xsl:value-of select="normalize-space(.)"/> -->
                <xsl:value-of select="."/>
                <xsl:text><![CDATA[</TD></TR>]]></xsl:text>
            </xsl:for-each>
            <xsl:text><![CDATA[
        <TR>
            <TD ALIGN="LEFT"><B>/Exit: </B></TD>
        </TR>
        ]]></xsl:text>
            <xsl:for-each select="onExit/line">
                <xsl:text><![CDATA[<TR><TD ALIGN="LEFT">]]></xsl:text>
                <!-- <xsl:value-of select="normalize-space(.)"/> -->
                <xsl:value-of select="."/>
                <xsl:text><![CDATA[</TD></TR>
        ]]></xsl:text>
            </xsl:for-each>
            <xsl:text><![CDATA[</TABLE>>];
]]></xsl:text>
        </xsl:for-each>
    <xsl:text>
    // guards
</xsl:text>

    <xsl:for-each select="
            //*[
                @CurrentStateTest
                    and not(@CurrentStateTest = '')
                    and not(@CurrentStateTest = 'InitPSubState')
                    and guard
                ]">
            <xsl:text>
    </xsl:text>
            <xsl:value-of select="@CurrentStateTest"/>
<xsl:text><![CDATA[ [shape=point, xlabel="]]></xsl:text>

            <xsl:for-each select="guard/line">
                <!-- <xsl:value-of select="normalize-space(.)"/> -->
                <xsl:value-of select="."/>
                <xsl:text>
</xsl:text>
            </xsl:for-each>
            <xsl:text><![CDATA["];
]]></xsl:text>
        </xsl:for-each>
        <xsl:text>

    // transitions
</xsl:text>

    <xsl:for-each select="//*[ @NextStateLabel ]">
        <xsl:value-of select="ancestor::*[@CurrentStateTest][1]/@CurrentStateTest"/
      >
        <xsl:text> -> </xsl:text>
        <xsl:value-of select="@NextStateLabel"/>
        <xsl:text><![CDATA[[label=<<TABLE BORDER="0" CELLBORDER="0">
    <TR><TD BORDER="1" SIDES="B">]]></xsl:text>
        <xsl:value-of select="@EventLabel"/>
        <xsl:text><![CDATA[</TD></TR>]]></xsl:text>
        <xsl:for-each select="onTransition/line">
            <xsl:text><![CDATA[
            <TR><TD ALIGN="LEFT">]]></xsl:text>
            <xsl:value-of select="."/>
            <xsl:text><![CDATA[</TD></TR>]]></xsl:text>
        </xsl:for-each>
        <xsl:text><![CDATA[
    </TABLE>>];
]]></xsl:text>
    </xsl:for-each>
    <xsl:text>
}</xsl:text>
    </xsl:template>

</xsl:stylesheet>
```

Fig. 3. XSLT to generate GraphViz diagram description from an annotated AST

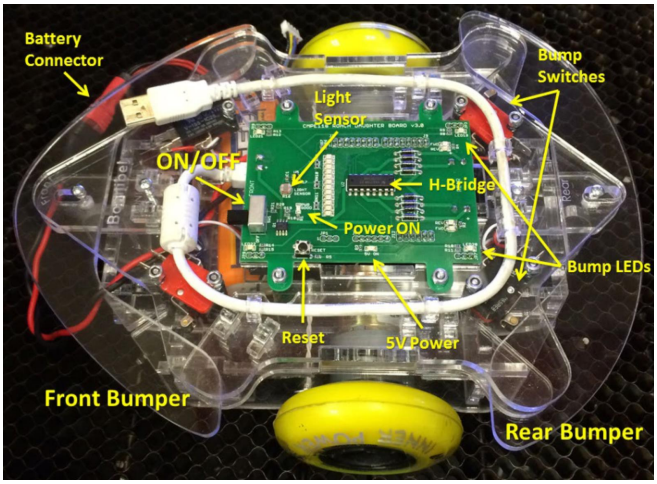

Fig. 4. A wheeled robot controlled by code in figure 5

IV. Discussion

A. AST XPATH

Initially, we employed regular expression patterns [12] for diagram generation data extraction. This method fell short as it treated source code linearly, struggling with nested structures like switch-default and if-elseif-else constructs.

To overcome these limitations, we shifted to using a C parser and Abstract Syntax Trees (ASTs). ASTs represent the hierarchical nature of source code, enabling us to use XPATH [13], a pattern language designed for tree structures. This approach is more effective than regular expressions for parsing nested code patterns.

Consider this XPATH used in our tool:

`ancestor::*[@CurrentStateTest][1]/@CurrentStateTest`

This XPATH works as follows:

- `ancestor::*[@CurrentStateTest][1]`: It locates the nearest ancestor element with a
- `CurrentStateTest` attribute in the AST hierarchy. The process involves:
  - `ancestor::*` to select all ancestor elements.
  - `[@CurrentStateTest]` to filter ancestors with the `CurrentStateTest` attribute.

```
1 ES_Event RunTemplateHSM(ES_Event ThisEvent) {
2    uint8_t makeTransition = FALSE; TemplateHSMState_t nextState; ES_Tattle();
3
4    switch (CurrentState) {
5    case InitPState:
6       if (ThisEvent.EventType == ES_INIT) {
7          InitLightSubHSM(); InitDarkSubHSM(); InitJigSubHSM(); ES_Timer_SetTimer(
                ↪ JIG_TIMER, JIG_TIME); nextState = InDark; makeTransition = TRUE;
                ↪  ThisEvent.EventType = ES_NO_EVENT;
8       }
9       break;
10
11   case InLight:
12      ThisEvent = RunLightSubHSM(ThisEvent);
13      switch (ThisEvent.EventType) {
14      case ES_ENTRY: ES_Timer_InitTimer(JIG_TIMER, JIG_TIME); break;
15      case ES_EXIT: ES_Timer_StopTimer(JIG_TIMER); break;
16      case LIGHT_TO_DARK: nextState = InDark; makeTransition = TRUE; ThisEvent.
              ↪ EventType = ES_NO_EVENT; break;
17      case ES_TIMEOUT: nextState = Jig; makeTransition = TRUE; ThisEvent.EventType =
              ↪  ES_NO_EVENT; ES_Timer_SetTimer(JIG_SPIN_TIMER, JIG_SPIN_TIME);
              ↪ break;
18      }
19      break;
20
21   case InDark:
22      ThisEvent = RunDarkSubHSM(ThisEvent);
23      switch (ThisEvent.EventType) {
24      case ES_ENTRY: StopMotors(); break;
25      case DARK_TO_LIGHT: nextState = InLight; makeTransition = TRUE; ThisEvent.
              ↪ EventType = ES_NO_EVENT; break;
26      }
27      break;
28
29   case Jig:
30      ThisEvent = RunJigSubHSM(ThisEvent);
31      switch (ThisEvent.EventType) {
32      case JIG_FINISHED: nextState = InLight; makeTransition = TRUE; ThisEvent.
              ↪ EventType = ES_NO_EVENT; break;
33      case LIGHT_TO_DARK: nextState = InDark; makeTransition = TRUE; ThisEvent.
              ↪ EventType = ES_NO_EVENT; break;
34      }
35      break;
36   }
37
38   if (makeTransition == TRUE) {
39      RunTemplateHSM(EXIT_EVENT); CurrentState = nextState; RunTemplateHSM(
              ↪ ENTRY_EVENT);
40   }
41
42   ES_Tail(); return ThisEvent;
43 }
```

Fig. 5. This FSM, representing the primary level in a hierarchical state machine (HSM), controls a wheeled robot modeled after a cockroach (shown in figure 4). It exhibits behaviors like moving in darkness and freezing in light, with an added periodic 'jig dance'.

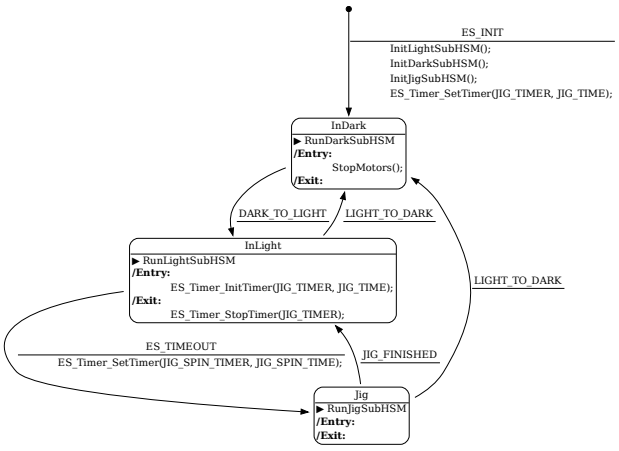

Fig. 6. State diagram generated from FSM code in figure 5

  - `[1]` to pick the first element from this filtered set.

- `/@CurrentStateTest`: Retrieves the `CurrentStateTest` attribute's value from the selected ancestor.

To walk the AST and fetch information as above is impractical with regular expressions.

B. Annotation Pipeline

Our second prototype attempted to directly convert ASTs into state diagrams. This was acceptable for simple diagrams however it soon proved overly complex and unmanageable, when adding features like event parameters, transition logic, and guards.

To address this, we developed a third prototype featuring an annotation pipeline. This pipeline breaks down the diagram generation process into distinct steps, each handling a specific type of annotation. This modular approach allows for easier debugging and verification of each step. After the annotations are complete, the AST is ready for a straightforward transformation into a state diagram using a single XSLT step. This final step uses the pre-annotated AST and three loops to fill out a predefined diagram description template as shown in figure 3.

At present, our annotation pipeline comprises fifteen XSLT steps (lines 41-55 in figure 1). Additional steps can be incorporated as needed for new diagram features or to handle more AST patterns. An example of one such early annotation step is illustrated in Figure 9. This step determines the diagram label associated with the current state and adds it as an attribute named `CurrentStateTest`.

Figure 9 includes an XPATH pattern that targets `block_items` AST elements based on specific criteria:

- `@class='Case' or @class='Default'`: This selects `block_items` nodes either with a `class` attribute value of `Case` or `Default`.
- `../../../block_items[@class='Switch']/cond[@class='ID' and @name='CurrentState']`: The process here is:
  - `../../..`: Ascends three levels in the AST from the current `block_items` node.
  - `/block_items[@class='Switch']`: Selects `block_items` nodes that are children of the node reached and have a `class` attribute of `Switch`.
  - `/cond[@class='ID' and @name='CurrentState']`: Then selects `cond` nodes that have a `class` attribute of `ID` and a `name` attribute of `CurrentState`.
- `and not(@CurrentStateTest)`: Excludes nodes already tagged with a `CurrentStateTest` attribute.

This XPATH pattern selects `block_items` nodes classified as either `Case` or `Default`, but only if they are hierarchically related to `block_items` nodes of class `Switch` with a child `cond` node meeting specific criteria (`class='ID'` and `name='CurrentState'`). These nodes must not already have a `CurrentStateTest` attribute. This ensures no overwriting if `CurrentStateTest` is already computed in another step.

The outcome of this XSLT is tagging all branches of switch statements conditional on the variable `CurrentState` with a `CurrentStateTest` attribute. This attribute holds the XPATH value referencing the label of the current switch branch:

```
./expr[@class='ID']/@name
```

The `CurrentStateTest` attribute's purpose is to track the current state label, allowing subsequent pipeline logic to reference this label without recalculating. If the current state is determined differently, like through if-elseif-else constructs

```
ES_Event RunHSM_Top_Orienting(ES_Event ThisEvent) {

uint8_t makeTransition=FALSE; HSM_Top_OrientingState_t nextState; ES_Event postEvent
      ↪ ; ES_Tattle(); uint8_t nextFromTrack; uint8_t nextFromTape;

switch (CurrentState) {
case InitPSubState:
   if (ThisEvent.EventType==ES_INIT) {
      wallHit=FALSE; barrierCount=0; barrierTrack=BARRIER_NULL; barrierTape=
            ↪ BARRIER_NULL; fieldSide=FIELD_UNKNOWN; centerTimerTime=
            ↪ TIMER_TICKS_CENTER_BUMP;
      turningTimerTime=DCMOTOR_TIME_TURN_90DEG; nextState=Find; makeTransition=TRUE;
            ↪  ThisEvent.EventType=ES_NO_EVENT;
   }
   break;

case Find:
   switch (ThisEvent.EventType) {
   case ES_ENTRY: DCMotor_Drive(DCMOTOR_DRIVE_SPEED, FORWARDS); break;
   case ES_EXIT: DCMotor_Stop(); break;
   case BUMPER_PRESSED: wallHit=TRUE; centerTimerTime=TIMER_TICKS_CENTER_BUMP;
         ↪ nextState=Align; makeTransition=TRUE; ThisEvent.EventType=ES_NO_EVENT;
         ↪  break;
   case TRACK_ENTERED: barrierTrack=barrierCount; centerTimerTime=
         ↪ TIMER_TICKS_CENTER_TRACK; nextState=Center; makeTransition=TRUE;
         ↪ ThisEvent.EventType=ES_NO_EVENT; break;
   case TAPE_ENTERED: barrierTape=barrierCount; centerTimerTime=
         ↪ TIMER_TICKS_CENTER_TAPE; nextState=Center; makeTransition=TRUE;
         ↪ ThisEvent.EventType=ES_NO_EVENT; break;
   }
   break;

case Align:
   switch (ThisEvent.EventType) {
   case ES_ENTRY: ES_Timer_InitTimer(TIMER_TOP_ORIENTING, TIMER_TICKS_ALIGN);
         ↪ DCMotor_Turn(DCMOTOR_DRIVE_SPEED, FORWARDS, LEFT); break;
   case ES_EXIT: DCMotor_Stop(); break;
   case ES_TIMEOUT:
      if (ThisEvent.EventParam==TIMER_TOP_ORIENTING) {
         nextState=Center; makeTransition=TRUE; ThisEvent.EventType=ES_NO_EVENT;
      }
      break;
   }
   break;

case Center:
   switch (ThisEvent.EventType) {
   case ES_ENTRY:
      ES_Timer_InitTimer(TIMER_TOP_ORIENTING, centerTimerTime); DCMotor_Drive(
            ↪ DCMOTOR_DRIVE_SPEED, BACKWARDS);
      if (wallHit==TRUE) barrierCount++; break;
   case ES_EXIT: DCMotor_Stop(); break;
   case ES_TIMEOUT:
      if (ThisEvent.EventParam==TIMER_TOP_ORIENTING) {
         if (barrierCount < BARRIER_COUNT) {
            nextState=Rotate;
         } else {
            if (barrierTrack==(BARRIER_COUNT - 1)) {
               nextFromTrack=0;
            } else if (barrierTrack==BARRIER_NULL) {
               nextFromTrack=BARRIER_NULL;
            } else { nextFromTrack=barrierTrack + 1; }
            if (barrierTape==(BARRIER_COUNT - 1)) {
               nextFromTape=0;
            } else if (barrierTape==BARRIER_NULL) {
               nextFromTape=BARRIER_NULL;
            } else { nextFromTape=barrierTape + 1; }
            if (barrierTrack==BARRIER_NULL) {
               fieldSide=FIELD_UNKNOWN;
            } else if (barrierTape==BARRIER_NULL) {
               fieldSide=FIELD_UNKNOWABLE;
            } else if (nextFromTrack==barrierTape) {
               fieldSide=FIELD_LEFT;
            } else if (nextFromTape==barrierTrack) {
               fieldSide=FIELD_RIGHT;
            }
            if ((fieldSide==FIELD_LEFT) || (fieldSide==FIELD_RIGHT) || (fieldSide==
                  ↪ FIELD_UNKNOWABLE)) {
               nextState=Turning_Beacon;
               turningTimerTime=DCMOTOR_TIME_TURN_90DEG * (barrierTrack + 1);
            } else {
               nextState=Turning_OtherSide; turningTimerTime=DCMOTOR_TIME_TURN_90DEG
                     ↪  * (barrierTape + 1); wallHit=FALSE; barrierCount=0;
                     ↪ barrierTrack=BARRIER_NULL; barrierTape=BARRIER_NULL;
                     ↪ fieldSide=FIELD_UNKNOWN; centerTimerTime=
                     ↪ TIMER_TICKS_CENTER_BUMP; turningTimerTime=
                     ↪ DCMOTOR_TIME_TURN_90DEG;
            }
         }
         makeTransition=TRUE; ThisEvent.EventType=ES_NO_EVENT;
      }
      break;
   }
   break;

case Rotate:
   switch (ThisEvent.EventType) {
   case ES_ENTRY: ES_Timer_InitTimer(TIMER_TOP_ORIENTING, TIMER_TICKS_ROTATE);
         ↪ DCMotor_TankTurn(DCMOTOR_TURN_SPEED, RIGHT); break;
   case ES_EXIT: DCMotor_Stop(); break;
   case ES_TIMEOUT:
      if (ThisEvent.EventParam==TIMER_TOP_ORIENTING) {
         nextState=Find; makeTransition=TRUE; ThisEvent.EventType=ES_NO_EVENT;
      }
      break;
   }
   break;

case Turning_Beacon:
   switch (ThisEvent.EventType) {
   case ES_ENTRY: ES_Timer_InitTimer(TIMER_TOP_ORIENTING, turningTimerTime);
         ↪ DCMotor_TankTurn(DCMOTOR_TURN_SPEED, RIGHT); break;
   case ES_EXIT: DCMotor_Stop(); break;
   case ES_TIMEOUT:
      if (ThisEvent.EventParam==TIMER_TOP_ORIENTING) {
         postEvent.EventType=ORIENTED; postEvent.EventParam=fieldSide; PostHSM_Top(
               ↪ postEvent); nextState=InitPSubState; makeTransition=TRUE;
               ↪ ThisEvent.EventType=ES_NO_EVENT;
      }
      break;
   }
   break;

case Turning_OtherSide:
   switch (ThisEvent.EventType) {
   case ES_ENTRY: ES_Timer_InitTimer(TIMER_TOP_ORIENTING, turningTimerTime);
         ↪ DCMotor_TankTurn(DCMOTOR_TURN_SPEED, RIGHT); break;
   case ES_EXIT: DCMotor_Stop(); break;
   case ES_TIMEOUT:
      if (ThisEvent.EventParam==TIMER_TOP_ORIENTING) {
         nextState=Driving_OtherSide; makeTransition=TRUE; ThisEvent.EventType=
               ↪ ES_NO_EVENT;
      }
      break;
   }
   break;

case Driving_OtherSide:
   switch (ThisEvent.EventType) {
   case ES_ENTRY: DCMotor_Drive(DCMOTOR_DRIVE_SPEED, FORWARDS); break;
   case ES_EXIT: DCMotor_Stop(); break;
   case TAPE_EXITED: nextState=Find; makeTransition=TRUE; ThisEvent.EventType=
         ↪ ES_NO_EVENT; break;
   }
   break;
}

if (makeTransition==TRUE) {
   RunHSM_Top_Orienting(EXIT_EVENT); CurrentState=nextState; RunHSM_Top_Orienting(
         ↪ ENTRY_EVENT);
}

ES_Tail(); return ThisEvent;
}
```

Fig. 7. A lower-level FSM in a multi-tiered HSM for a competition robot.

instead of a switch statement, another template can handle that scenario. Hence, the downstream logic needing the current state label does not depend on the specific logic computing the `CurrentStateTest` attribute.

C. Limitations and Challenges

Some limitations and challenges associated with our tool include:

a) CPP Includes: In Section II-A4, we discuss the application of CPP to generate a C code stream independent of other files. The success of CPP hinges on accessing all necessary project and library include files. Although our prototype includes standard files, version mismatches with users' codes may necessitate manual updates to the CPP launch command. To facilitate this, our prototype outputs each CPP command, allowing users to modify the CPP launch command as needed if the default setting fails.

b) AST Understanding: The AST's complexity compared to the original source code is evident in Figure 11, which depicts the AST for the first branch of a CurrentState switch statement from Figure 10. The AST's verbosity and size—often expanding a few hundred lines of code into thousands—pose significant navigational challenges.

c) Annotation Development: Understanding the effects of annotation steps requires examining the AST before and after each step. Figure 12 demonstrates the use of `tee` commands for capturing AST states around the `s00400_add_CascadeElements.xml` annotation step. Differences can be highlighted using

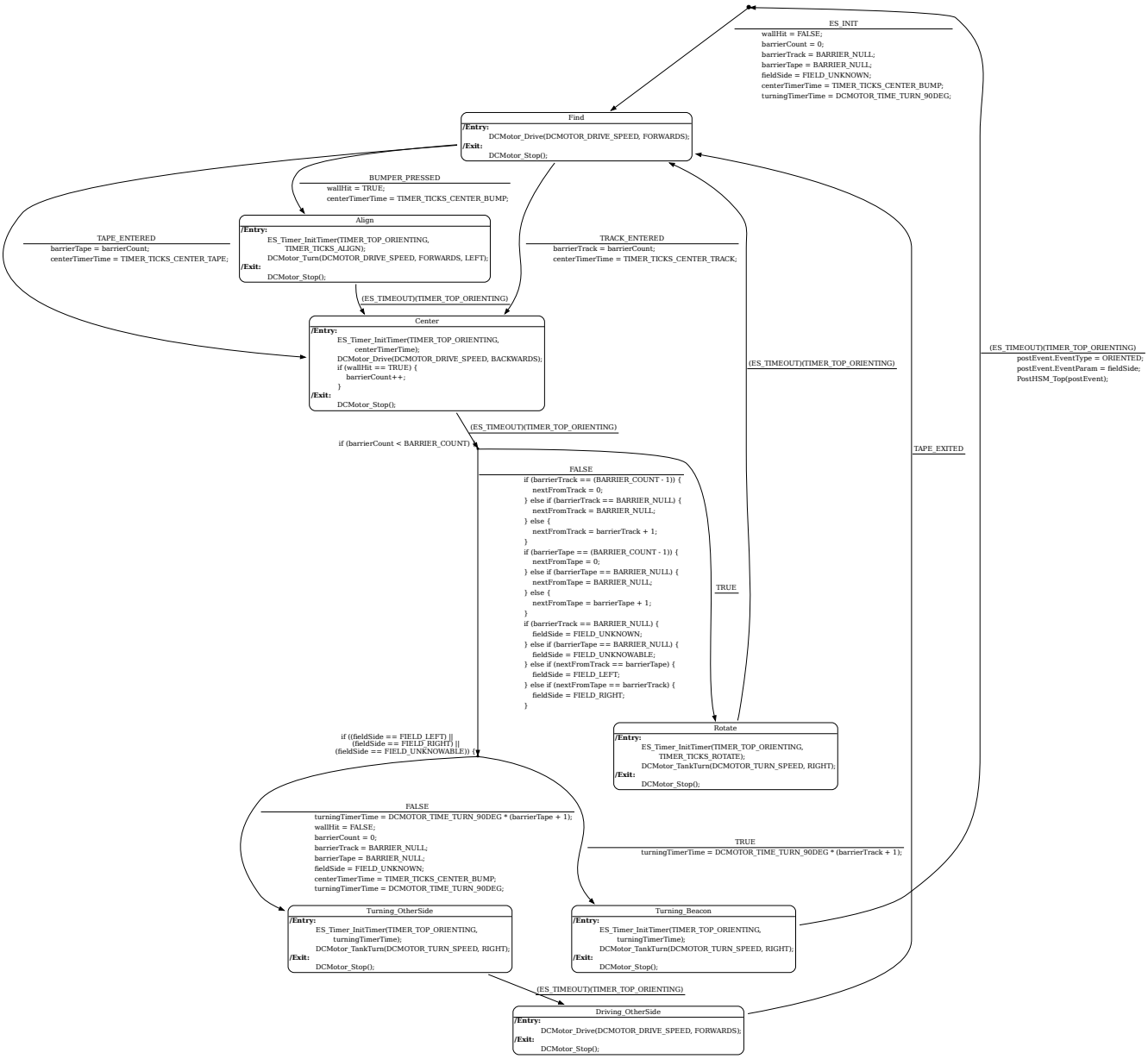

Fig. 8. State diagram generated from FSM code in figure 7

`diff -u before.xml after.xml`
or an IDE's equivalent function.

D. Future Work

Our objectives include:

- **Expanding FSM Code Patterns:** As UCSC students increasingly utilize our prototype, supporting a wider range of FSM code patterns remains our primary focus.
- **Enhancing Configuration and Compatibility:** Plans to broaden the prototype's appeal beyond UCSC involve making naming conventions configurable and extending support to FSMs in Java, Python, JavaScript, etc. We also aim to generate diagrams in formats like Mermaid.js and PlantUML, and introduce new diagram types such as Harel Statecharts and Activity Diagrams.
- **Static Code Analysis for FSM Diagnostics:** Future enhancements include static code analysis to identify FSM programming errors, like states with incomplete transitions or potential deadlocks, where the FSM could freeze without any viable transitions.
- **Open Source Collaboration:** To foster collaborative development and wider adoption, the complete prototype is available under an Open Source license (AGPLv3) and can be accessed free of charge at: https://github.com/jlesner/smv2.

```
    <xsl:template match="
       block_items[
           (@class='Case'
               or @class='Default')
           and (
               ../../..
               /block_items[@class='Switch']
               /cond[@class='ID' and @name='CurrentState']
           )
           and not(@CurrentStateTest)
       ]">
       <xsl:copy>
           <xsl:apply-templates select="@*"/>
           <xsl:attribute name="CurrentStateTest">
               <xsl:value-of select="./expr[@class='ID']/@name"/>
           </xsl:attribute>
           <xsl:apply-templates select="node()"/>
       </xsl:copy>
   </xsl:template>
```

Fig. 9. Example annotation that adds `CurrentStateTest` attribute

```
switch (CurrentState) {
   case InitPSubState:
      if (ThisEvent.EventType == ES_INIT)
      {
         ES_Timer_StopTimer(TIMER_TOP_RELOADING);
         trackCrossings = 0;
         nextState = Turning;
         makeTransition = TRUE;
         ThisEvent.EventType = ES_NO_EVENT;
      }
      break;
      ...
```

Fig. 10. Sample code snip showing just the first case in a switch statement

```
<block_items class="Switch" line="602">
   <cond class="ID" line="602" name="CurrentState"/>
   <stmt class="Compound" line="602">
      <block_items class="Case" line="603">
         <expr class="ID" line="603" name="InitPSubState"/>
         <stmts class="If" line="604">
            <cond class="BinaryOp" line="604" op="==">
               <left class="StructRef" line="604" type=".">
                  <name class="ID" line="604" name="ThisEvent"/>
                  <field class="ID" line="604" name="EventType"/>
               </left>
               <right class="ID" line="604" name="ES_INIT"/>
            </cond>
            <iftrue class="Compound" line="605">
               <block_items class="FuncCall" line="606">
                  <name class="ID" line="606" name="ES_Timer_StopTimer"/>
                  <args class="ExprList" line="606">
                     <exprs class="ID" line="606" name="TIMER_TOP_RELOADING"/>
                  </args>
               </block_items>
               <block_items class="Assignment" line="607" op="=">
                  <lvalue class="ID" line="607" name="trackCrossings"/>
                  <rvalue class="Constant" line="607" type="int" value="0"/>
               </block_items>
               <block_items class="Assignment" line="608" op="=">
                  <lvalue class="ID" line="608" name="nextState"/>
                  <rvalue class="ID" line="608" name="Turning"/>
               </block_items>
               <block_items class="Assignment" line="609" op="=">
                  <lvalue class="ID" line="609" name="makeTransition"/>
                  <rvalue class="ID" line="609" name="TRUE"/>
               </block_items>
               <block_items class="Assignment" line="610" op="=">
                  <lvalue class="StructRef" line="610" type=".">
                     <name class="ID" line="610" name="ThisEvent"/>
                     <field class="ID" line="610" name="EventType"/>
                  </lvalue>
                  <rvalue class="ID" line="610" name="ES_NO_EVENT"/>
               </block_items>
            </iftrue>
         </stmts>
         <stmts class="Break" line="612"/>
      </block_items>
      ...
```

Fig. 11. Sample AST snip matching code in figure 10

```
find "$src_path" -name '*.c.cp5' \
| while read f ; do
    echo "visualizing '$f'"
    (
        sx=saxonb-xslt
        cat "$f" \
        | tr -d '\r' \
        | ( egrep -avi '^[[:blank:]]*$|^#|va_list|__attribute__' || true ) \
        | perl -pe's{__extension__}{ }g; s{__}{}g; ' \
        | python3 c_ast_xml.py \
        | tee "${f}.xml" \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00005_identity.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00100_declutter_attributes.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00200_add_bLine_eLine.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00300_add_CurrentStateTest.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00300_add_EventParamTest.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00300_add_EventTypeTest.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00300_add_NextStateLabel.xml \
        | tee before.xml  \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00400_add_CascadeElements.xml \
        | tee after.xml  \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00500_add_CascadeLabel.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00550_add_EventLabel.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00560_add_Guard_Element.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00570_add_Guard_Attributes.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00600_add_onEntry_onExit.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00600_add_onTransition2.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00620_drop_unwanted_code.xml \
        | $sx -s:/dev/stdin -o:/dev/stdout -xsl:s00800_gv_digraph4.xml \
        | perl -pe 's/ && / &amp;&amp; /g;
                    s/ < / &lt; /g;
                    s/ > / &gt; /g;
                    s/ <= / &lt;= /g;
                    s/ >= / &gt;= /g;
                  ' \
        > "${f}.gv"

        dot -Tpng "${f}.gv" -o "${f}.png"
        dot -Tpdf "${f}.gv" -o "${f}.pdf"
        dot -Tsvg "${f}.gv" -o "${f}.svg"

    ) 2>&1
done
```

Fig. 12. Lines 19 and 21 show how annotation AST captures are done

V. Conclusion

We have described a new tool for automatically creating visualizations of Finite State Machines (FSMs), which is particularly useful in software engineering and robotics. The tool simplifies the creation of state diagrams, which is usually complex and error-prone, especially for intricate FSMs. It uses naming conventions, Abstract Syntax Tree (AST) patterns, and XSLT transformations to generate accurate FSM visuals from the source code, accommodating various coding patterns. This not only saves time and reduces errors but also helps in understanding FSM structures, proving especially beneficial in educational settings like UCSC's mechatronics courses.

The tool's ability to handle different FSM code patterns, including hierarchical state machines and transition guards, shows its versatility. It is being used in education to help students learn and implement FSMs in robotics. Although it currently works in a specific programming environment and with certain naming conventions, there's potential for expanding its capabilities to more programming languages, diagram types, and FSM verification diagnostics.

In summary, this tool marks a significant advancement in automating state diagram generation, improving the design and debugging of FSMs in various applications, especially in education.

Acknowledgments

This project was initiated and funded by CAHSI Undergraduates Program and supported by National Science Foundation Grants #2034030 and #1834620. Christopher Lesner helped with UNIX commands. Bailen Lawson supplied the FSM code samples used for AST pipeline development and testing. ChatGPT assisted with Latex and polished our paragraphs. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of CAHSI or the National Science Foundation.

APPENDIX A
README.MD

The State Machine Visualizer (SMV) is a tool for visualizing the structure and behavior of state machines in your code. Follow these steps to set up and use the tool.

Step-by-Step Instructions

STEP 1: Download the Script: First, download the `smv.bash` script using the following command:

```
wget https://raw.githubusercontent.com/jlesner/smv2/main/smv.bash
```

STEP 2: Script Review and Preparation: Before running the script, it's important to understand its functions:

- **Inspect Changes:** Review the `smv.bash` script to understand the changes it will make. It installs necessary tools like git, curl, and podman if they are not already present on your system.
- **Password Prompt:** The script uses `sudo apt-get`, which might prompt you for your password to install missing tools.
- **First-Time Setup:** On its initial run, `smv.bash` will download the latest version of the State Machine Visualizer and install required dependencies.
- **System Requirements:** The script is designed for Linux systems with the apt package manager, such as Ubuntu. Windows users can use Ubuntu/WSL2, and macOS users might need to run Ubuntu in a VM.
- **Containerization:** To create a suitable environment, `smv.bash` builds a Linux container, installing additional dependencies (Python, Java, etc.) and executes the SMV code within this container. Note that this container requires approximately 900MB of space.
- **Cleanup:** At the end of the script, instructions are provided to remove the installations made by `smv.bash`. These instructions are for when you are done using SMV and want to remove it. Leaving things installed allows `smv.bash` to run faster.

STEP 3: Running the Script: To run the State Machine Visualizer, use the following command, replacing `${path_to_code}` with the path to your state machine files:

```
bash smv.bash ${path_to_code}
```

STEP 4: Viewing the Results: After running the script, you can find the generated `.gv` and `.png` files, which are the visual representations of your state machines, using this command:

```
find ${path_to_code} -name '*.cp5*'
```

By following these steps, you should be able to successfully set up and use the State Machine Visualizer for your projects.

APPENDIX B
SMV.BASH

This script sets up and runs a our tool on a folder containing *.c files, generating diagrams in *.gv and *.png and other formats. It handles dependencies, converts paths for Unix compatibility, verifies directories, fetches necessary files, and clones a Git repository if needed, with robust error handling and debugging options.

```
# This script carries out the setup and execution of state machine visualizer.
#
# This script is intended to be run on debian or ubuntu. It may work on other unix distributions but this is not tested.
#
# This script handles dependencies, environment setup, and execution within a Docker container.
#
# This script is designed to be robust and user-friendly, providing clear error messages and automatically installing missing components.
#
# This script launches smv_gen_png.bash which creates state machine diagrams (as *.gv and *.png files) from *.c files


# **Environment Variables**
#
# Environment variables specify folders this script uses:
#     `src_path` is for the top folder containing the source files to be visualized
#     `smv_path` is the home of the state machine visualizer tool
#     `pic32mx_include_path` for Microchip PIC32MX include files
#     `course_include_path` for course-specific include files
#
# src_path may be a relative path in unix or windows path format (e.g., /mnt/c/dev/ECE118 or C:\dev\ECE118)
# IMPORTANT: The rest must be absolute paths in unix path format (e.g., /mnt/c/dev/ECE118/include)
#



# **Example how to run this script**
#
# To use the default values for environment variables and download any needed pic32mx and course include files:
#
#     bash smv.bash $src_path
#
# If you already have pic32mx and course include files or want to use different versions specify their location as follows:
#
# (
#     export smv_path=~/smv # assuming you already have smv repo cloned here
#     && export pic32mx_include_path='/mnt/c/Program Files/Microchip/xc32/v4.10/pic32mx/include'
#     && export course_include_path='/mnt/c/dev/ECE118/include'
#     && bash smv.bash $src_path
# )
#



# **Debug Options** (Disabled by default)
#
# - These options are for debugging purposes and are commented out. If enabled, they provide trace and debugging information.
#
# set -o errtrace # This option, also known as -E, causes any trap on ERR to be inherited by shell functions, command substitutions, and commands executed in a subshell environment. The ERR trap is a mechanism in Bash that allows a function to be executed whenever a command exits with a non-zero status (indicating failure). With errtrace enabled, this behavior is extended to more parts of the script, making it easier to detect and handle errors.
# set -o functrace # This option enables function tracing in the script. It makes the DEBUG and RETURN traps (which are normally only triggered by the script itself) also be triggered by shell functions. The DEBUG trap typically runs before each command in the script, and the RETURN trap runs each time a shell function or a script executed with the . or source commands finishes executing. This option is useful for tracing the flow of execution through functions in a script.
# set -o xtrace # This option, often referred to as -x, is used for debugging purposes. It prints each command and its arguments to the standard error (stderr) before executing it. This trace includes expansions of variables and commands, providing a detailed view of what's happening in the script. It's particularly useful for seeing the flow of execution and understanding how data is being manipulated.
# export SHELLOPTS # This command exports the SHELLOPTS variable, making it an environment variable that is inherited by child processes. SHELLOPTS is a special shell variable that contains a colon-separated list of enabled shell options.


# **Argument Parsing**
#
# Check if an argument is passed to the script (arg="$1"). If not, print an error message and exit.

arg="$1"

if [[ -z "$arg" ]]; then
    echo "Error: You must supply src_path as an argument."
    exit 1
fi


# **Script Safety Options** (Enabled by default)
#
set -o nounset # Causes the script to exit if an uninitialized variable is used. Helps catch mistakes.
set -o pipefail # Causes a pipeline (e.g., cmd1 | cmd2) to return the exit status of the last command in the pipe that failed.
set -o errexit # Exits the script if any command fails (returns a non-zero status). Together with pipefail this stops script on first error.


# **Apply UNIX Path Conversion**
#
# Check if the argument contains a colon (:), suggesting a Windows-style path. If so, convert it to a Unix path using wslpath. Otherwise, use the argument directly.
#
if [[ "$arg" == *:* ]]; then
  unix_path="$(echo ${arg} | tr "\n" "\0" | xargs -0 wslpath -u )"
else
  unix_path="$arg"
fi

```

```
# **Directory Validation**
#
# Verify if the supplied unix_path argument is a directory. If not, it print an error message and exit.
#
if [[ ! -d "$unix_path" ]]; then
    echo "Error: Supplied argument src_path must be a folder."
    exit 1
fi


# **Apply Absolute Path Conversion**
#
# If supplied unix_path argument is relative convert it to be absolute using pwd. (e.g., local path ECE118 may be mapped to /mnt/c/dev/ECE118)
# This is needed to docker to work properly.
#

export src_path=`cd "${unix_path}" ; pwd`


# **Dependency Checks and Installations**
#
# Check for the existence of various tools (curl, git, docker) and try to install them if they are missing.
#
# NOTE: sudo below will prompt user for password if the user is not already root. This is needed to install packages.

( apt-get --version 2>&1 ) >/dev/null \
    || ( \
        echo "Error: apt-get package manager missing; please use debian or ubuntu to run me."
        exit 1
    )

( curl --version 2>&1 ) >/dev/null \
    || ( \
        sudo apt-get update \
            && sudo apt install -y curl
    )

( git --version 2>&1 ) >/dev/null \
    || ( \
    sudo apt-get update \
        && sudo apt install -y git
    )

( docker --version 2>&1 ) >/dev/null \
    || ( \
        sudo apt-get update \
            && sudo apt-get install -y podman podman-docker
            # podman emulates docker and takes less resources; to install docker run
            # apt-get install -y docker.io
    )


# **Set Default Values for Variables**:
#
# Default values for `smv_path`, `src_path`, `pic32mx_include_path`, and `course_include_path` are set using using parameter expansion.
# The bash parameter expansion `:-` operator assigns a default value if the variable is unset or null.
#
export smv_path="${smv_path:-$HOME/smv}"
export dep_path="${dep_path:-$HOME/smv_dep}"
export course_include_path="${course_include_path:-${dep_path}/ECE118/include}"
export pic32mx_include_path="${pic32mx_include_path:-${dep_path}/pic32mx/include}"


# **Fetch Dependencies**
#
# For course_include_path and pic32mx_include_path, checks if these directories exist.
# If not, fetch these from specified URLs using curl and extract them.
#

if [[ -d "${course_include_path}" ]]; then
    echo "course_include_path exists: ${course_include_path}"
else
    echo "course_include_path does not exist: ${course_include_path}"
    echo "fetching from http://www.ufafu.com/smv/ECE118.tgz"
    (
        mkdir -p "${course_include_path}"
        cd "${course_include_path}"/../..
        curl -L http://www.ufafu.com/smv/ECE118.tgz \
            | tar -xzf -
    )
fi


if [[ -d "${pic32mx_include_path}" ]]; then
    echo "pic32mx_include_path exists: ${pic32mx_include_path}"
else
    echo "pic32mx_include_path does not exist: ${pic32mx_include_path}"
    echo "fetching from http://www.ufafu.com/smv/pic32mx.tgz"
    (
        mkdir -p "${pic32mx_include_path}"
        cd "${pic32mx_include_path}"/../..
        curl -L http://www.ufafu.com/smv/pic32mx.tgz \
            | tar -xzf -
    )
fi


# **Clone Repository**
#
# If the smv_path directory doesn't exist, it clone a Git repository from a specified URL.
#

if [[ -d "${smv_path}" ]]; then
    echo "smv_path exists: ${smv_path}"
else
    echo "smv_path does not exist: ${smv_path}"
    echo "cloning repo from https://github.com/jlesner/smv2"
    (
        base_smv=$(basename "${smv_path}")
 dirname_smv=$(dirname "${smv_path}")
        cd "${dirname_smv}"
        git clone https://github.com/jlesner/smv2 ${base_smv}
    )
fi


# **Build and Launch Container**
#
# Build a container image tagged smv:0.05 and then launch a new temporary container with this image.
# Mount several volumes from the host to the container and execute smv_gen_png.bash inside the container.
#

(
    cd "${smv_path}"
    ( docker images smv:0.05 \
            | tail -n 1 \
            | grep ^REPOSITORY \
      && docker build -t smv:0.05 .
    ) || true
)

docker run \
    --rm \
    -it \
    -v "${smv_path}":"${smv_path}" \
    -v "${src_path}":"${src_path}" \
    -v "${pic32mx_include_path}":"${pic32mx_include_path}" \
    -v "${course_include_path}":"${course_include_path}" \
    smv:0.05 \
    bash -c \
        " \
        cd '${smv_path}' \
            && export smv_path='${smv_path}' \
            && export src_path='${src_path}' \
            && export pic32mx_include_path='${pic32mx_include_path}' \
            && export course_include_path='${course_include_path}' \
            && bash ./smv_gen_png.bash \
        "


# **Uninstallation**
#
# Explain how to uninstall showing commands to remove Docker images, apt packages, and directories related to the installation.
#
echo "
NOTE: To uninstall smv, the following three commands may (or may not) be useful:
    docker rmi smv:0.05 ubuntu:20.04
    sudo apt remove podman podman-docker # first check you no longer need this
    rm -rf '${smv_path}' '${dep_path}' # first check these were actually used for your installation!
"
```

Appendix C
Dockerfile

This Dockerfile shows the instructions our tool uses build a custom environment container image. It specifies the base operating system, libraries, and dependencies, as well as the application code to be included. This allows for the creation of a lightweight, portable, and consistent environment across different machines and platforms.

```
FROM ubuntu:20.04

LABEL maintainer="jlesner@ucsc.edu"

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y \
    python3 \
    python3-pip \
    libsaxonb-java \
    graphviz \
    cpp \
    curl \
    dos2unix \
    default-jre-headless \
    wget

RUN ln -s /usr/bin/python3 /usr/bin/python

RUN pip3 install lxml==4.9.2 numpy==1.24.4 pycparser==2.21
```

Appendix D
smv_png_gen.bash

This script processes C source files to create state machine diagrams by first applying a C preprocessor (CPP) to handle macros and include directives, then using Python and XSLT transformations to generate an abstract syntax tree (AST) and

refine it for visualization. Finally, it employs GraphViz to produce state diagram visualizations in PNG format from the prepared AST.

```
# This script creates state machine diagrams (as *.gv and *.png files) from *.c files

# **Environment Variables**
#
# Environment variables specify folders this script uses:
#     `src_path` is for the top folder containing the source files to be visualized
#     `smv_path` is the home of the state machine visualizer tool
#     `pic32mx_include_path` for Microchip PIC32MX include files
#     `course_include_path` for course-specific include files
#
# Example how to launch this script:
#
# (
#     cd ${smv_path}
#     && export src_path= # path to state machine *.c files (inside this folder or children)
#     && export smv_path= # path to your local copy of state machine visualizer aka smv repo
#     && export pic32mx_include_path= # path to pic32mx include files ( eg Microchip/xc32/v4.10/pic32mx/include )
#     && export course_include_path= # path to course include files ( eg ECE118 )
#     && bash ./smv_gen_png.bash
# )


# **Script Safety Options** (Enabled by default)
#
set -o nounset # Causes the script to exit if an uninitialized variable is used. Helps catch mistakes.
set -o pipefail # Causes a pipeline (e.g., cmd1 | cmd2) to return the exit status of the last command in the pipe that failed.
set -o errexit # Exits the script if any command fails (returns a non-zero status). Together with pipefail this stops script on first error.

# **Debug Options** (Disabled by default)
#
# - These options are for debugging purposes and are commented out. If enabled, they provide trace and debugging information.
#
# set -o errtrace # This option, also known as -E, causes any trap on ERR to be inherited by shell functions, command substitutions, and commands executed in a subshell environment. The ERR trap is a mechanism in Bash that allows a function to be executed whenever a command exits with a non-zero status ( indicating failure). With errtrace enabled, this behavior is extended to more parts of the script, making it easier to detect and handle errors.
# set -o functrace # This option enables function tracing in the script. It makes the DEBUG and RETURN traps (which are normally only triggered by the script itself) also be triggered by shell functions. The DEBUG trap typically runs before each command in the script, and the RETURN trap runs each time a shell function or a script executed with the . or source commands finishes executing. This option is useful for tracing the flow of execution through functions in a script.
# set -o xtrace # This option, often referred to as -x, is used for debugging purposes. It prints each command and its arguments to the standard error ( stderr) before executing it. This trace includes expansions of variables and commands, providing a detailed view of what's happening in the script. It's particularly useful for seeing the flow of execution and understanding how data is being manipulated.
# export SHELLOPTS # This command exports the SHELLOPTS variable, making it an environment variable that is inherited by child processes. SHELLOPTS is a special shell variable that contains a colon-separated list of enabled shell options.


# **Setting Default Values for Variables**:
#
# Default values for `smv_path`, `src_path`, `pic32mx_include_path`, and `course_include_path` are set using using parameter expansion.
# The bash parameter expansion `:-` operator assigns a default value if the variable is unset or null.

smv_path="${smv_path:-$PWD}"

cd "${smv_path}" # Change the current working directory to the one specified in `smv_path`.

src_path="${src_path:-$PWD/samples/ECE118_RoachLab_Bailen}"
src_path="${src_path:-$PWD/samples/ECE218_Team1_F2022}"

pic32mx_include_path="${pic32mx_include_path:-$HOME/smv_dep/pic32mx/include}"
course_include_path="${course_include_path:-$HOME/smv_dep/ECE118/include}"


# **CPP Macro Encoding (`encode.pl`)**:
#
# `epath`` points to a *runtime* generated Perl script which encodes #define macros to prevent them beign expanded by CPP so that diagrams have labels like TURN_RIGHT instead of 0x12345678

epath="$src_path"/encode.pl

find \
    "${src_path}" \
    "${course_include_path}" \
    -type f \( -name '*.h' -o -name '*.hpp' -o -name '*.c' \) \
    | tr "\n" "\0" \
    | xargs -0 cat \
    | dos2unix \
    | ( egrep -ai '^#define' || true ) \
    | perl -pe 's/#define (\w)(\w+)[ \(].*$/s{\\b$1$2\\b}{$1zz0912819zz$2}g; # encode123 /g;' \
    | ( grep encode123 || true ) \
    | perl -pe 's/ # encode123//g;' \
    | sort | uniq \
    > "${epath}"

# Step-by-step:
#
# 1. **Finding Files (`find` Command)**:
#     - The `find` command searches for files within the directories specified by `${src_path}` and `${course_include_path}`.
#     - The `-type f` option restricts the search to files (as opposed to directories or other types of items).
#     - The `-name` options specify the file extensions to look for: `*.h`, `*.hpp`, and `*.c`, which are typically C and C++ header and source files.
#
# 2. **Replacing Newlines (`tr` Command)**:
#     - The `tr "\n" "\0"` command translates newline characters (`\n`) into null characters (`\0`). This is often done to handle filenames that contain spaces or unusual characters safely.
#
# 3. **Concatenating Files (`xargs` and `cat` Commands)**:
#     - The `xargs -0 cat` part reads the null-terminated strings from the previous command and uses `cat` to concatenate the contents of the files.
#
# 4. **Converting Line Endings (`dos2unix` Command)**:
#     - The `dos2unix` command converts Windows line endings (CRLF) to Unix line endings (LF), ensuring compatibility in Unix/Linux environments.
#
# 5. **Extracting `#define` Directives (`egrep` Command)**:
#     - The `egrep -ai '^#define'` command extracts lines that start with `#define`, ignoring case (`-i`). The `|| true` ensures that the pipeline doesn't fail if `egrep` doesn't find any matching lines.
#
# 6. **Perl Regular Expression Processing**:
#     - Two Perl (`perl -pe`) commands are used to perform regular expression substitutions on the extracted lines:
#         - The first `perl` command encodes certain patterns found after `#define` directives.
#             It is targeting macro names and replacing parts of them with a unique string (`zz0912819zz`), marked with a comment `# encode123` for later identification.
#         - The second `perl` command removes the `# encode123` marker, leaving only the modified macro names.
#
# 7. **Filtering and Deduplicating (`grep`, `sort`, `uniq`)**:
#     - The `grep encode123 || true` command filters the lines containing the `encode123` marker.
#     - The `sort | uniq` commands sort the results and remove duplicate lines.
#
# 8. **Redirecting Output**:
#     - Finally, the output of this pipeline is redirected to a file specified by the `${epath}` variable.


# **Include List (for CPP) **
#
# Next we build `ilist` a space-separated string of include paths, each prefixed with `-I`.
# This format is used by CPP to specify directories where the it will look for header files.
# If there are include directories at `path1/include` and `path2/include`, `ilist` will end up looking something like `-I'path1/include' -I'path2/include'`.

ilist=$( \
       find "$src_path" \
           -type d \
           -name include \
           -print0 \
       | xargs -0 -I{} echo "-I'{}'" \
       | tr "\n" " " \
   )

# Step-by-step:
#
# 1. **`find "$src_path"`**:
#     - The `find` command is used to search through directories and files. In this case, it's looking within the directory specified by the `src_path` variable.
#
# 2. **`-type d`**:
#     - This option tells `find` to look only for directories (`d`).
#
# 3. **`-name include`**:
#     - This option restricts the search to directories named `include`.
#
# 4. **`-print0`**:
#     - This outputs the found directory names, with each name terminated by a null character (`\0`) instead of a newline.
#       This is useful for handling filenames that contain spaces, newlines, or other unusual characters.
#
# 5. **`| xargs -0 -I{} echo "-I'{}'"`**:
#     - The output from `find` is piped (`|`) to `xargs`, which is used to build and execute command lines from the input.
#     - `-0` tells `xargs` to expect null-terminated inputs (which matches the output of `find ... -print0`).
#     - `-I{}` is a placeholder that will be replaced by each input line in the command `echo "-I'{}'"`.
#     - The `echo` command outputs each directory path prefixed with `-I'`, which is a common way to specify include directories for compilers.
#
# 6. **`| tr "\n" " "`**:
#     - This translates (using the `tr` command) all newline characters into spaces.
#       This is important because `xargs` by default outputs items separated by newlines, but the intention here is to create a space-separated list.


# **Include Configure List (for CPP)**:
#
# Here we build `iconfig2`, a space-separated string of include flags for each directory containing an `ES_Configure.h` file.

iconfig2=$( \
       find "$src_path" \
           -type f \
           -name 'ES_Configure.h' \
           -print0 \
       | xargs -0 -I{} dirname {} \
       | tr "\n" "\0" \
       | xargs -0 -I{} echo "-I'{}'" \
       | tr "\n" " " \
```

```
    )

# Step-by-step:
#
# 1. **`find "$src_path" -type f -name 'ES_Configure.h' -print0`**:
#     - This command searches within the directory specified by `src_path` for files (`-type f`) named `ES_Configure.h`.
#     - The `-print0` option prints the full file path followed by a null character (`\0`). This is useful for handling filenames with spaces or unusual characters.
#
# 2. **`| xargs -0 -I{} dirname {}`**:
#     - The output from `find` is piped (`|`) into `xargs`, which executes the `dirname` command for each found file path.
#     - `xargs -0` tells `xargs` to expect null-terminated input (matching the `-print0` from `find`), which is safer for handling filenames with special characters.
#     - `dirname {}` extracts the directory path of each found file, with `{}` being a placeholder for each input line.
#
# 3. **`| tr "\n" "\0"`**:
#     - This translates (`tr`) newline characters (`\n`) into null characters (`\0`), preparing the list of directories for another round of `xargs`.
#
# 4. **`| xargs -0 -I{} echo "-I'{}'"`**:
#     - Here, `xargs` processes each null-terminated string (directory path) and echoes it with `-I` prepended and surrounded by single quotes.
#     - This step formats each directory path into a format suitable for inclusion flags (e.g., `-I'/path/to/dir'`) used to specify directories where include files are located.
#
# 5. **`| tr "\n" " "`**:
#     - Finally, this translates newline characters into spaces, converting the multi-line output into a single line.


# **Apply CPP**
#
# Here we apply CPP to C source files that contain references to (`nextState`) as these are deemed to contain state machines.
#

find "$src_path" -type f -name '*.c' -print0 \
    | xargs -0 egrep -l nextState \
    | while read f ; do
        ff="`basename \"${f}\"`"
        b="`dirname \"${f}\"`"
        (
            cd "$b" \
            && echo "amalgamating '${f}'" \
            && cat "${ff}" \
                | dos2unix \
                | perl -p "${epath}" \
                | ( egrep -avi '^#define '  || true ) \
                > "${ff}.undef" \
            && echo "( cd '$b' ; cpp \
                -I\"${course_include_path}\" \
                -I\"${pic32mx_include_path}\" \
                $ilist \
                $iconfig2 \
                -I'${b}' \
                -I. \
                '${ff}.undef' \
                )" \
                | tee /dev/stderr \
                | bash \
                | perl -pe '
                    s{zz0912819zz}{}g;
                ' \
                | dos2unix \
                > "${ff}.cp5"
                # \
            # && rm -f "${ff}.undef"
                # -I'$iconfig' \
        ) 2>&1
        # | tee "${f}.log"
    done

rm -f "$epath" # encode.pl script has served its purpose and is no longer needed

# Step-by-step:
#
# 1. **Finding Files and Identifying Relevant Ones**:
#     - `find "$src_path" -type f -name '*.c' -print0`: This command finds all C source files (`*.c`) in the directory specified
#       by `src_path`. The `-print0` option outputs the file names separated by null characters, which is useful for handling filenames with spaces.
#     - `| xargs -0 egrep -l nextState`: The file paths are piped to `egrep` to search for the pattern `nextState` in these files.
#       The `-l` option makes `egrep` list only the names of files where the pattern is found.
#
# 2. **Processing Each File**:
#     - The script then enters a `while read f` loop to process each file that contains the `nextState` pattern.
#     - `ff="`basename \"${f}\"`"`: Extracts the filename from the full path.
#     - `b="`dirname \"${f}\"`"`: Extracts the directory path from the full path.
#
# 3. **Amalgamation and Preprocessing**:
#     - The script changes directory to the file's directory (`cd "$b"`) and performs a series of operations:
#         - It echoes a message indicating the start of processing for the file.
#         - The file is concatenated (`cat "${ff}"`), converted from DOS to UNIX text format (`dos2unix`), and then processed with a Perl script (`perl -p "${epath}"`).
#         - The perl ${epath} script is generated above and protects macros from being expanded by CPP.
#         - Any line starting with `#define` is removed using `egrep -avi '^#define '`, and `|| true` ensures that the pipeline does not fail if `egrep` doesn't match any lines.
#         - The processed content is saved into a temporary file (`"${ff}.undef"`).
#
# 4. **Further Processing with C Preprocessor**:
#     - The script constructs a command to run the C preprocessor (`cpp`) on the `.undef` file, including various include paths
#       (specified by `course_include_path`, `pic32mx_include_path`, `ilist`, `iconfig2`, and the current and root directories).
#     - This command is echoed (and logged via `tee /dev/stderr`) and then executed in a subshell (`| bash`).
#
# 5. **Post-Processing and Final Output**:
#     - Output from the C preprocessor is further processed with Perl (`perl -pe`), replacing `zz0912819zz` with nothing.
#     - The purpose of this is to remove ${epath} encoding that protects macros from being expanded by CPP.
#     - The final output is converted again to UNIX format (`dos2unix`) and saved as `"${ff}.cp5"`.
#
# 6. **Cleanup and Logging**:
#     - Commented-out lines (`# && rm -f "${ff}.undef"` and `# | tee "${f}.log"`) show cleanup and logging which are currently disabled.
#



# **Apply PycParser and XSLT and GraphViz**
#
# We locate `*.c.cp5` files in `src_path` (generated above) and build their abstract syntax tree AST using PycParser,
# then apply pipeline of XLST templates, and finally use GraphViz to generate state diagram in PNG format.

find "$src_path" -name '*.c.cp5' \
    | while read f ; do
        echo "visualizing '$f'"
        (
            cat "$f" \
                | tr -d '\r' \
                | ( egrep -avi '^[[:blank:]]*$|^#|va_list|__attribute__' || true ) \
                | perl -pe's{__extension__}{ }g; s{__}{}g; ' \
                | python3 c_ast_xml.py \
                | tee "${f}.xml" \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00005_identity.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00100_declutter_attributes.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00200_add_bLine_eLine.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00300_add_CurrentStateTest.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00300_add_EventParamTest.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00300_add_EventTypeTest.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00300_add_NextStateLabel.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00400_add_CascadeElements.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00500_add_CascadeLabel.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00550_add_EventLabel.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00560_add_Guard_Element.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00570_add_Guard_Attributes.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00600_add_onEntry_onExit.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00600_add_onTransition2.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00620_drop_unwanted_code.xml \
                | saxonb-xslt -s:/dev/stdin -o:/dev/stdout -xsl:xslt/s00800_gv_digraph4.xml \
                | perl -pe 's/ && / &amp;&amp; /g;
                            s/ < / &lt; /g;
                            s/ > / &gt; /g;
                            s/ <= / &lt;= /g;
                            s/ >= / &gt;= /g;
                        ' \
                > "${f}.gv"

            dot -Tpng "${f}.gv" -o "${f}.png"

        ) 2>&1
        # | tee "${f}.log"
    done

# Step-by-step:
#
# 1. **Finding Files and Iteration**:
#    - The `find` command locates all files with the `.c.cp5` extension within `src_path`.
#    - The `while read f` loop processes each found file one by one.
#
# 2. **Initial Processing of Each File**:
#    - Each file's contents are read and echoed with `cat "$f"`.
#    - The `tr -d '\r'` command removes carriage return characters, which is useful for ensuring compatibility with Unix line endings.
#    - A series of `egrep` filters out lines that are either blank, start with `#`, or contain specific strings like `va_list` or `__attribute__`.
#      The `|| true` ensures that the pipeline doesn't break if `egrep` doesn't find a match.
#
# 3. **Perl Script Processing**:
#    - The Perl one-liner makes two substitutions: it replaces `__extension__` with a space and removes double underscores (`__`).
#      The purpose of this is to ensure compatibility with the C parser used in the next step.
#
# 4. **Generating XML Representation**:
#    - The script uses `python3 c_ast_xml.py` to convert the processed C code into an XML representation of its abstract syntax tree (AST).
#
# 5. **Multiple XSLT Transformations**:
#    - The XML output is then piped through a series of XSLT (eXtensible Stylesheet Language Transformations) using `saxonb-xslt`.
#      Each transformation (`xslt/s00005_identity.xml`, etc.) progressively modifies the XML, to prepare it for visualization.
```

```
#      For example the purpose of s00005_identity.xml is to format the XML output of PyParser to allow diff to work better during debugging.
#      For example the purpose of s00100_declutter_attributes.xml remove AST elements not needed for subsequent processing.
#      See comments inside each XSLT *.xml template for more details.
#
# 6. **HTML Escape Processing**:
#    - A final Perl script further processes the GraphViz diagram description, replacing certain logical and comparison operators
#      (`&&`, `>`, `<`, `<=`, `>=`) with their HTML entity equivalents to ensure proper parsing by GraphViz.
#      NOTE more HTML escapes may be needed such as:
#         s/&/&amp;/g;
#         s/"/&quot;/g;
#         s/'\''/&apos;/g;
#
# 7. **GraphViz Visualization**:
#    - The processed output is saved as a GraphViz file (`${f}.gv`).
#    - The `dot` command from GraphViz is then used to generate a PNG image from the `.gv` file, visualizing the structure of the C code.
#
# 8. **Error Handling and Logging**:
#    - The `2>&1` notation combines standard output and error streams, which can be used for logging or debugging (as indicated by the commented out `| tee "${f}.log"`).


# **Cleanup Intermedite Files**
#
# Find all files within `src_path` that end with `.c.cp5` or `.c.cp5.xml`, and then safely and forcefully delete them.
# The use of null characters as delimiters in `xargs` makes this command robust against file names with unusual characters or spaces.

find "$src_path" | egrep '\.c\.cp5$|\.c\.cp5\.xml$'  | tr "\n" "\0" | xargs -0 rm -f

# Step-by-step:
#
# 1. **`find "$src_path"`**:
#    - This command searches for all files and directories within the directory specified by the variable `src_path`.
#
# 2. **`egrep '\.c\.cp5$|\.c\.cp5\.xml$'`**:
#    - The output from `find` is piped to `egrep`, which is a version of `grep` used for pattern matching with regular expressions.
#    - The regex `'\.c\.cp5$|\.c\.cp5\.xml$'` is used to filter the list of files. It looks for files that end with `.c.cp5` or `.c.cp5.xml`. The `$` ensures that the pattern matches the end of the file name.
#
# 3. **`tr "\n" "\0"`**:
#    - This translates (or replaces) newline characters (`\n`) in the output with null characters (`\0`).
#      This is done because file names can potentially contain spaces or other special characters, which might be misinterpreted by the next command. Using null characters as delimiters avoids this issue.
#
# 4. **`xargs -0 rm -f`**:
#    - The modified output is then piped to `xargs`, which builds and executes command lines from standard input.
#    - The `-0` option tells `xargs` to expect input items to be terminated by a null character, which matches the output from the `tr` command.
#    - `rm -f` is the command that `xargs` executes. `rm` is the remove command in Unix/Linux, and the `-f` option forces deletion without prompting for confirmation, even if the files are write-protected.
```

REFERENCES

[1] Graphviz. [Online]. Available: https://graphviz.org/

[2] Mermaid. [Online]. Available: https://mermaid.js.org/

[3] Plantuml. [Online]. Available: https://plantuml.com/

[4] D. van Heesch. Doxygen. [Online]. Available: https://www.doxygen.nl/

[5] O. M. Group. Unified modeling language specification. [Online]. Available: https://www.omg.org/spec/UML/

[6] S. Systems. Enterprise architect. [Online]. Available: https://sparxsystems.com/products/ea/index.html

[7] W. W. W. Consortium. Extensible stylesheet language transformations (xslt) version 3.0. [Online]. Available: https://www.w3.org/TR/xslt-30/

[8] Mplab® x integrated development environment (ide). [Online]. Available: https://www.microchip.com/en-us/tools-resources/develop/mplab-x-ide

[9] Mplab® xc compilers. [Online]. Available: https://www.microchip.com/en-us/tools-resources/develop/mplab-xc-compilers

[10] E. Bendersky and contributors. pycparser. [Online]. Available: https://github.com/eliben/pycparser

[11] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley Professional, 2013, comprehensive guide on C++ by its original creator.

[12] Perl regular expressions. [Online]. Available: https://perldoc.perl.org/perlre

[13] W. W. W. Consortium. Xml path language (xpath) specification. [Online]. Available: https://www.w3.org/TR/xpath/