An Automated Tool for State Machine Diagram Generation in Mechatronics Education

Jasmine Lesner[1] and Gabriel Hugh Elkaim[2]

***Abstract*— We present a targeted tool designed to address a specific challenge in teaching event-driven control systems: the automated visualization of State Machines directly from source code. While State Machines are fundamental in event-driven control systems, their growing complexity can make debugging and maintaining correspondence between design and code increasingly difficult for students. Our tool employs lightweight static analysis techniques leveraging Abstract Syntax Tree patterns to automatically generate accurate state diagrams from student implementations. By focusing on the immediate needs of our students and the constraints of our educational setting, this tool improves State Machine design, debugging, and maintenance within our curriculum. Anecdotal evidence suggests that students benefit directly from this accessible and frequently usable solution, enhancing their understanding and workflow in complex, state-driven systems. This approach demonstrates the value of developing targeted educational tools that address specific classroom needs, even within the broader context of existing research in the field.**

I. Introduction

State machines are crucial in software engineering, especially in embedded systems and robotics. Their implementation often involves complex structures of conditional statements that check current states, events, parameters, and guard conditions. This complexity, along with the varied coding patterns used by different programmers, presents a significant challenge when teaching event-driven control systems: **How can we automatically generate accurate visual representations of state machines directly from source code?**

To address this challenge, we developed a tool to reverse-engineer and visualize state machines by analyzing source code. Our approach uses Abstract Syntax Tree (AST) analysis and an annotation pipeline implemented using XPATH [1] and XSLT [2], designed to be lightweight and accessible to students.

The tool is tailored for a mechatronics capstone course at the University of California, Santa Cruz (UCSC) [3] which is based on principles outlined in [4]. This project-based course combines lab exercises with open-ended team projects, where students build and program their robots (see Figure 1) to compete in public events. State diagrams are essential for helping students visually understand, design, and communicate their robots' behavior.

A key simplifying assumption of our approach is that it is designed for *explicit state machines* where the state is tracked using a single *current state* variable. Our approach does not apply to *implicit state machines*, which have state distributed across multiple variables and are best handled with user-guided extraction [5]. Our simplifying assumption aligns with the mechatronics programming style taught at UCSC and allowed us to build a state diagram visualization tool that is entirely automatic.

While we acknowledge the existence of broader solutions in the research community, our tool is specifically designed to be immediately accessible and beneficial to UCSC students allowing for frequent use and rapid feedback. Anecdotal evidence suggests that students have already benefited from this tailored approach.

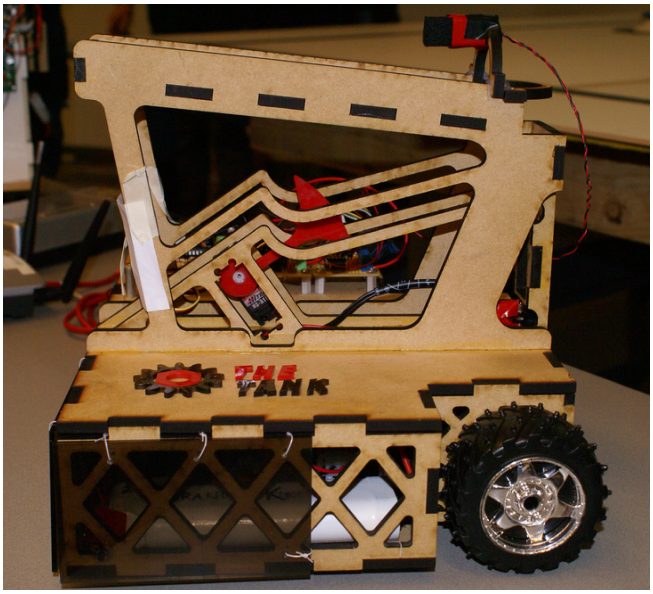

Fig. 1. Student built UCSC mechatronics competition robot.

A. Related Work

State machine visualization is a well-researched area with various approaches and tools available. However, many existing solutions are either too complex, unavailable, or unsuitable for our specific context: supporting UCSC's mechatronics engineering students.

- **Dynamic Analysis** tools observe software behavior in real-time to infer state machines [6], [7], [8]. While these provide accurate representations of executed paths, they are often too complex for frequent use in an educational setting and may miss inactive code paths.

[1]Jasmine Lesner is with the Computer Science department at University of California, Santa Barbara jlesner@ucsb.edu

[2]Gabriel Hugh Elkaim is with Computer Engineering department at the University of California, Santa Cruz elkaim@soe.ucsc.edu

- **Static Analysis** approaches examine source code without execution [9], [10], [11], [12]. These can be more comprehensive but can produce overly complex diagrams [13] which are challenging to interpret.

Our tool employs static analysis, chosen for its ability to provide comprehensive coverage of all potential states and transitions. This is particularly crucial for our students during the development and debugging phases of new robots, where visualizing rare or impossible states might reveal programming mistakes common among novice programmers.

While powerful tools and research projects exist in this domain, they present barriers for our specific use case:

- Visualization tools like Graphviz [14], Mermaid.JS [15], and PlantUML [16] require manual creation of diagram descriptions, which is error prone and time-consuming for students.
- Documentation generators like Doxygen [17] need source code annotations, adding another avenue for errors to students' workflows.
- UML tools such as Enterprise Architect [18] and research tools such as [19], [20] and [12] are too complex or unavailable for regular student use.

These limitations highlighted the need for a tool specifically designed for mechatronics at UCSC, capable of automatically creating state machine diagrams directly from the students' source code, without requiring embedded annotations or interactive user intervention. Our solution addresses this need by providing a lightweight, accessible tool that understands the event-driven software design patterns used at UCSC.

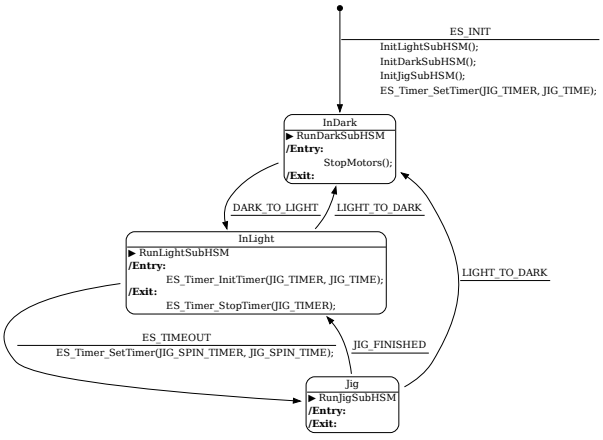

Fig. 2. State diagram generated from source code shown in Figure 3. This diagram shows the top level of a HSM where each state (`InDark`, `InLight`, `Jig`) runs a nested EFSM (omitted for brevity).

B. Contributions

We present a tool that automates the creation of state diagrams from source code, making the process efficient and error-free. Our key contributions are:

```c
ES_Event RunTemplateHSM(ES_Event ThisEvent) {
   uint8_t makeTransition = FALSE; TemplateHSMState_t
        ↪ nextState; ES_Tattle();
   switch (CurrentState) {
   case InitPState:
      if (ThisEvent.EventType == ES_INIT) {
         InitLightSubHSM(); InitDarkSubHSM();
              ↪ InitJigSubHSM(); ES_Timer_SetTimer(
              ↪ JIG_TIMER, JIG_TIME);
         nextState = InDark;
         makeTransition = TRUE;
         ThisEvent.EventType = ES_NO_EVENT;
      }
      break;
   case InLight:
      ThisEvent = RunLightSubHSM(ThisEvent);
      switch (ThisEvent.EventType) {
         case ES_ENTRY: ES_Timer_InitTimer(JIG_TIMER,
              ↪ JIG_TIME); break;
         case ES_EXIT: ES_Timer_StopTimer(JIG_TIMER);
              ↪ break;
         case LIGHT_TO_DARK: nextState = InDark;
              ↪ makeTransition = TRUE; ThisEvent.
              ↪ EventType = ES_NO_EVENT; break;
         case ES_TIMEOUT: nextState = Jig; makeTransition
              ↪ = TRUE; ThisEvent.EventType = ES_NO_EVENT
              ↪ ; ES_Timer_SetTimer(JIG_SPIN_TIMER,
              ↪ JIG_SPIN_TIME); break;
      }
      break;
   case InDark:
      ThisEvent = RunDarkSubHSM(ThisEvent);
      switch (ThisEvent.EventType) {
         case ES_ENTRY: StopMotors(); break;
         case DARK_TO_LIGHT: nextState = InLight;
              ↪ makeTransition = TRUE; ThisEvent.
              ↪ EventType = ES_NO_EVENT; break;
      }
      break;
   case Jig:
      ThisEvent = RunJigSubHSM(ThisEvent);
      switch (ThisEvent.EventType) {
         case JIG_FINISHED: nextState = InLight;
              ↪ makeTransition = TRUE; ThisEvent.
              ↪ EventType = ES_NO_EVENT; break;
         case LIGHT_TO_DARK: nextState = InDark;
              ↪ makeTransition = TRUE; ThisEvent.
              ↪ EventType = ES_NO_EVENT; break;
      }
      break;
   }
   if (makeTransition == TRUE) {
      RunTemplateHSM(EXIT_EVENT);
      CurrentState = nextState; RunTemplateHSM(ENTRY_EVENT
           ↪ );
   }
   ES_Tail(); return ThisEvent;
}
```

Fig. 3. Source code implementing the top level of a HSM which controls a robot modeled after a cockroach. The robot moves in darkness, freezes in the presence of light and has a periodic 'jig dance'.

1) **Support for FSM, EFSM, and HFSM diagrams.** We use pycparser [21], a complete C language parser and our approach can be extended to other languages like C++, Java, and C# by replacing pycparser with srcML [22].
2) **Handling complex code structures**[1]. State machine behavior is often implemented through various nested control structures: `if-else` branches and `switch-case` statements that check guard conditions and manage state transitions. Our tool supports a wide range of these structures, including `switch-case` groups with-

[1]Our technical report [23] details supported code patterns.

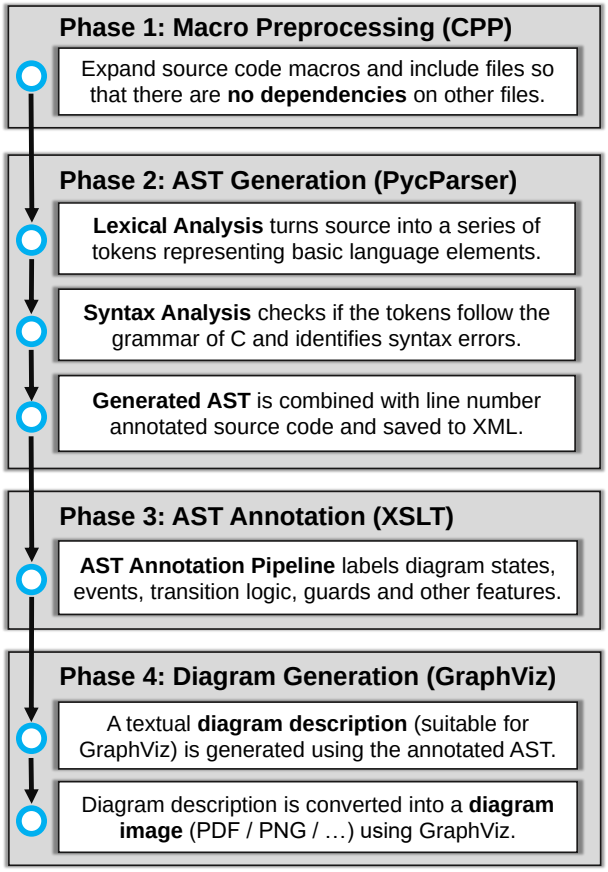

Fig. 4. Automatic diagram generation has four phases.

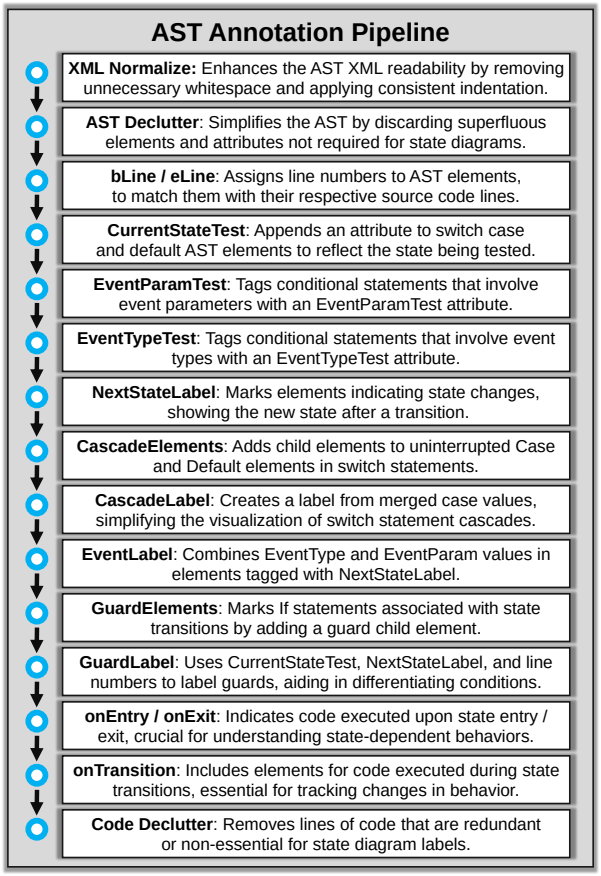

Fig. 5. Pipeline of AST annotations discussed in section II-C.

out `break` statements also fall through to `default` clauses.

As the **first open-source tool**[2] **to automatically generate state diagrams directly from source code**, we aim to foster broader use and collaboration.

II. Method

Figure 2 displays the state diagram generated from the source code in Figure 3. This EFSM is the top level in a HSM which controls a wheeled robot modeled after a cockroach. The HSM controls behaviors like moving in darkness, freezing in the presence of light and a periodic 'jig dance'. Each top level HSM state runs a nested EFSM, which are omitted for brevity. The key state diagram features include: **Initial State**: Represented by a black dot at the top, marks the HSM's starting point. The `CurrentState` is assigned to this state before the machine begins. **States**: Depicted as rounded boxes with the state name at the top. These may contain entry and exit logic and form the top level of the `CurrentState` switch statement. In HSMs, states can incorporate sub-states or reference other state machines, indicated by an arrowhead pointing to the sub-state function name. **Transitions**: Depicted as arrows between states and labeled with triggering events (above line) and executed code (below line). Events can be triggered by external inputs or internal timers. Transitions are established by assigning the `nextState` variable. **Guards**: Are conditional statements mid-transition that determine which transition to take (including returning to the original state), enabling fine-grained control over state changes. Our tool generates state diagrams (like Figure 2) directly from source code (like Figure 3) by operating in four phases (Figure 4):

A. Phase 1: Macro Preprocessing with CPP

Before its AST can be generated we must first preprocess source code to expand any macros contained such as `#include` directives so the source code our parser (PycParser) reads has no dependencies on other files. To enhance diagram clarity we prevent expansion of macros defined by students (as opposed to macros defined by whatever micro-controller framework they are using) so generated diagrams have human friendly labels (e.g. `LEFT_BUMBER`) instead of cryptic numeric codes (e.g. `0x02F4`).

B. Phase 2: AST Generation

We employ PycParser [21] to parse the self-contained source code, generating an XML structure comprising 'ast' and 'code' subtrees. The 'code' subtree holds the original source code lines annotated with line numbers and the 'ast' subtree contains the parsed Abstract Syntax Tree (AST), each

[2]Source code: `https://github.com/jlesner/smv2` [24]

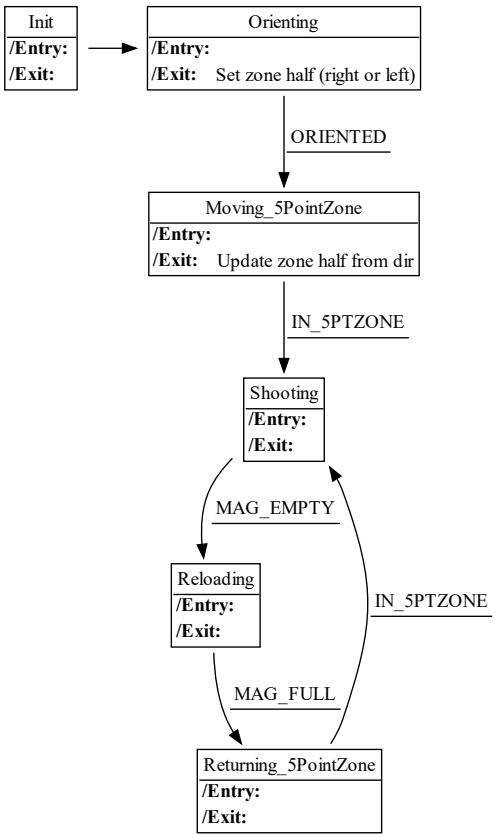

Fig. 6. This state diagram was manually created by a student team (using GraphViz) to represent the intended behavior of their robot for a UCSC mechatronics competition. The diagram shows the top level of an HSM that has an autonomous robot orient itself, move to a ball shooting zone, perform a shooting action sequence until its ball magazine is empty at which point it carries out a ball reloading action sequence until its ball magazine is full, thereafter returning to the ball shooting zone for more shooting.

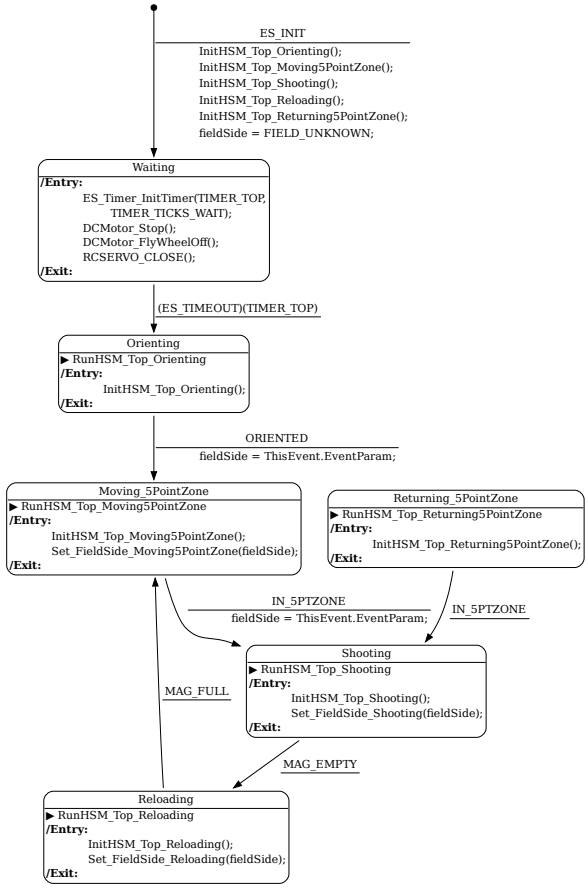

Fig. 7. Matching Figure 6 this diagram was automatically generated by our tool and represents the actual robot behavior as implemented by source code. When compared we can see how the implemented robot behavior deviates: (1) there is a new `Waiting` state which was added before the `Orienting` state likely to facilitate handling the robot after power up (2) the `Returning-5PointZone` state has no entry transitions which is likely an implementation mistake.

element of which references its matching source code lines so that these lines can be used to create diagram labels.

C. Phase 3: AST Annotation

Next we apply a pipeline of annotations as shown in Figure 5. In this pipeline we simplify the AST by removing unneeded elements and attributes. We mark significant parts of the AST, like switch cases and conditional statements, with attributes that indicate their roles in state transitions. Next, we label key elements that depict state changes and handle complex conditions within switch statements. Elements related to state transitions are also tagged to show changes in behavior and conditions affecting these transitions. Finally, we streamline label annotations by removing redundancies and prioritizing essential information.

D. Phase 4: Diagram Generation

Now the annotated AST is used to generate a GraphViz description of a diagram as follows: **Diagram Setup**: Output format is set to plain text suitable for Graphviz, and the entry point for the state machine is identified. **Loop over States and Guards**: Loop through AST elements representing different states and guards, excluding the entry state and guard conditions. Format these with suitable GraphViz styles and labels, including `onEntry` and `onExit` code blocks. **Loop over Transitions**: Loop through state transitions, adding them to the diagram description with their `onTransition` code blocks. Finally, the diagram description is rendered visually in formats that Graphviz supports (PNG, PDF, etc.).

III. User Evaluation

Our tool was evaluated by third parties (UCSC mechatronics capstone course teams) whose code was not used for development of our tool. Participants were emailed tool use instructions ('README' from [24]) and asked to try the tool and share their experiences. While our sample size is small, the *verbatim user comments* below provide insights.

Regarding **ease of use**, those familiar with Linux found the tool easy to use: *"It worked perfectly with 2 commands."*

However, some struggled and expressed a desire *"to have a GUI to script the commands in a more entry level manner for someone that is more unfamiliar with using command line jargon."*

Regarding **time savings**: *"Very helpful if you needed to visualize a more complex state machine and see what each state is doing. But if you just need a basic overview I think you could easily draw one out in about the same time with an iPad."* Another participant said, *"This tool was actually extremely time efficient. I have used other auto generators in the past and have found the setup for the files to be cumbersome. This would save me 90% of the time to set up alternatives and 95% of the time to draw it out manually."*

Regarding **accuracy**: *"Diagrams did seem very close to how they were meant to be implemented. The only discrepancies would be how the text overlaps some transitions so it's hard to tell exactly where things are going."* Another participant noted, *"Surprisingly accurate for the state machine we built. It followed the logical path with the conditions and outputs quite well. I cannot pinpoint any direct errors of logical fault within the first 10 minutes of looking at it."*

Additionally the UCSC mechatronics course instructor commented that *"the automated state machine diagrams demonstrated flaws in the students' implementations that were difficult to determine from looking at their source code but immediately obvious looking at the generated diagram. As a diagnostic tool, this is excellent to improve instruction to emphasise the correct coding patterns and where the students tend to get it wrong."*

To illustrate this point a state diagram manually created by a team using GraphViz is shown in Figure 6. This diagram was thought to represent the top level of an HSM that controls a robot. The robot orients itself, moves to a ball shooting zone, performs a shooting sequence until its ball magazine is empty, then reloads the magazine before returning to the ball shooting zone to repeat the process.

When the same robot's top-level HSM was analyzed by our tool, the resulting diagram (Figure 7): (1) Includes additional details, such as the names of the HSM's lower-level EFSMs, like `RunHSM_Top_Orienting` and `RunHSM_Top_Moving5PointZone`. (2) Displays a new `Waiting` state added before the `Orienting` state, possibly to facilitate handling the robot after power-up (3) Shows that the `Returning-5PointZone` state has no entry transitions, which appears to be an implementation error. Such errors are easier to spot using generated state diagrams than by inspecting code.

IV. Benchmark Evaluation

Our tool is typically used with HSMs, which have one top-level state diagram and one diagram per nested FSM so each tool run produces multiple diagrams. Since the generation of one diagram is independent of the others, we explored whether concurrent diagram generation could reduce user wait times. A benchmark was conducted to assess this potential benefit.

The tests were performed on a system running WSL2 Ubuntu 22.04.4 LTS Linux on Windows 10 Pro with an Intel Core i7-8850H CPU. This processor has six physical cores and twelve hyper-threaded virtual cores (vCores), each running between 800 MHz and 4200 MHz.

TABLE I
Benchmark Results

Run	vCore Usage	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

Table I shows the results of four benchmark runs. Each run processed identical code (∼70 files, ∼14,000 lines) to generate thirteen state diagrams – two of which are shown in Figures 2 and 7. The tests were conducted on a laptop connected to AC power, using the default power settings in Windows 10 Pro. Three virtual cores were occupied with background tasks, leaving nine cores available for benchmarking.

The benchmark results reveals the following:

1) **Diagram Generation Time:** Each state diagram takes less than ten seconds to generate, with a 20-25% variation in time between the fastest and slowest runs. While diagram complexity may play a role, this variation is likely due to CPU thermal throttling.
2) **Elapsed Time vs. vCore Usage:** Higher vCore usage did not lead to faster completion times. In fact, the longest processing times occurred with the highest vCore usage, again suggesting thermal throttling reduced core performance, increasing overall time despite higher vCore percentages.

Since our tool efficiently uses eight of the nine available vCores, there is limited potential for further parallelization. While servers with more vCores could benefit from concurrent diagram generation, typical laptops and PCs are unlikely to see significant performance gains from additional parallel processing.

V. Development

Initially, we used regular expressions [25] to extract details for state diagrams. This method struggled with varied code structures like `switch-case` and `if-elseif-else` constructs. To address this, we switched to a C parser and ASTs in XML, which allowed us to use XPATH and XSLT.

Our second approach transformed ASTs directly into state diagrams. This worked for simpler diagrams but became too complex for diagrams with event parameters, transition logic, and guards. We then developed a modular annotation pipeline (Figure 5) which breaks down diagram generation into steps each targeting a specific annotation type. For example Figure 8 shows one of the ways the CurrentStateTest (fourth from top in Figure 5) is implemented for `switch` statements. An alternative implementations is used for `if-elseif-else` constructs.

This new modular approach simplifies debugging and verification of each individual annotation step and the AST, once annotated, is then converted into a Graphviz state diagram description with a single XSLT template.

Currently, our tool uses XSLT, Perl, Python, and standard Unix commands within a Docker [26] container for Unix-like systems. Essential libraries include PycParser [21] for parsing source code and lxml [27] and libsaxonb-java [28] for XML processing, enhancing compatibility across different environments. This setup is well-suited for CI/CD pipelines and can be adjusted through environment variables, ensuring consistent operation across systems.

```
<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. 8. Implementation of the CurrentStateTest step in Figure 5. This demonstrates how branches of `switch` statements conditional on the variable `CurrentState` are tagged with a `CurrentStateTest` attribute. This attribute labels the current `switch` branch, enabling subsequent annotation pipeline logic to reference this label without recalculating it. The approach is designed to be modular, allowing for other templates to be triggered because the FSM current state may be determined differently, such as through `if-elseif-else` construct instead of a `switch` statement. Section IV in our technical report [23] explains how the XPATH / XSLT code above operates.

VI. Limitations and Future Work

As mentioned earlier a key simplifying assumption of our approach is that it is designed for *explicit state machines* where the state is tracked using a single *current state* variable. Our approach does not apply to *implicit state machines*, which have state distributed across multiple variables and are best handled with user-guided extraction [5].

Our approach also depends on consistent naming conventions to identify the current state, next state transitions, event types, and event parameters. While the specific variable names do not matter, they must be used consistently. Source code with inconsistent naming (i.e., using different names for the same thing) must be fixed before our tool can work effectively.

Looking ahead, state diagrams can visually display issues like incomplete transitions and states lacking exit transitions (deadlocks). Our tool could automatically detect and flag these problems in generated diagrams. We also aim to enhance our tool to generate diagram descriptions for visualization tools like Mermaid.js and PlantUML, and to add diagram types such as Harel Statecharts [29]. For large state machines, our tool could generate interactive diagrams that allow users to zoom in or drill down for more information.

VII. Conclusion

This paper presented a targeted tool designed to address a specific challenge in teaching event-driven control systems: the automated visualization of State Machines directly from source code. Our key contributions include:

1) Development of a lightweight, accessible tool that automatically generates accurate state diagrams from student implementations using static analysis techniques and Abstract Syntax Tree patterns.
2) Support for Finite State Machines (FSM), Extended Finite State Machines (EFSM), and Hierarchical State Machines (HSM) diagrams, with the ability to handle complex code structures.

Our approach demonstrates the value of developing targeted educational tools that address specific classroom needs, even within the broader context of existing research in the field. By focusing on the immediate needs of our students and the constraints of our educational setting, this tool improves the State Machine design, debugging, and maintenance processes within our curriculum.

The limitations of our approach include its design for explicit state machines tracked by a single current state variable and dependence on consistent naming conventions. Future work could involve automatic detection and flagging of issues like incomplete transitions and deadlocks, support for additional diagram types such as Harel Statecharts, and generation of interactive diagrams for large state machines.

This research contributes to the field of engineering education by providing an open-source tool that bridges the gap between theoretical understanding and practical implementation of state machines. It highlights the potential for targeted, domain-specific tools to enhance the learning experience in complex technical subjects.

Acknowledgments

We want to thank Bailen Lawson who supplied the code samples we used for tool development and testing. We also want to thank Ryan Taylor and Aidan Doshier and Julio Galan for assisting with evaluations. This project was initiated and funded by CAHSI Undergraduates Program and supported by National Science Foundation Grants #2034030 and #1834620.

References

[1] World Wide Web Consortium, "XML Path Language (XPATH) Specification." [Online]. Available: https://www.w3.org/TR/xpath/
[2] ——, "Extensible Stylesheet Language Transformations (XSLT) Version 3.0." [Online]. Available: https://www.w3.org/TR/xslt-30/
[3] G. H. Elkaim, "A hole in one: A project-based class on mechatronics," in *2011 IEEE International Conference on Microelectronic Systems Education*, 2011, pp. 35–38.
[4] J. E. Carryer, "The design of laboratory experiments and projects for mechatronics courses," *Mechatronics*, vol. 5, no. 7, pp. 787–797, 1995.

[5] W. Said, J. Quante, and R. Koschke, "Mining understandable state machine models from embedded code," *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 4759–4804, 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09865-0

[6] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Empirical Software Engineering*, vol. 21, no. 3, pp. 811–853, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9367-7

[7] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 501–510.

[8] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," *ACM Sigplan Notices*, vol. 37, no. 1, pp. 4–16, 2002.

[9] N. Walkinshaw, K. Bogdanov, S. Ali, and M. Holcombe, "Automated discovery of state transitions and their functions in source code," *Software Testing, Verification and Reliability*, vol. 18, no. 2, pp. 99–121, 2008.

[10] C. Mosler, "Reengineering of State Machines in Telecommunication Systems," *Department of Computer Science 3, RWTH Aachen University*, 2007.

[11] R. Knor, G. Trausmuth, and J. Weidl, "Reengineering c/c++ source code by transforming state machines," in *Development and Evolution of Software Architectures for Product Families: Second International ESPRIT ARES Workshop Las Palmas de Gran Canaria, Spain February 26–27, 1998 Proceedings 2*. Springer, 1998, pp. 97–105.

[12] M. Abadi and Y. A. Feldman, "Automatic recovery of statecharts from procedural code," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 238–241. [Online]. Available: https://doi.org/10.1145/2351676.2351711

[13] D. H. A. Zeeland, "Reverse-engineering state machine diagrams from legacy C-code," Ph.D. dissertation, Eindhoven University of Technology, 2009.

[14] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, *Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 127–148. [Online]. Available: https://doi.org/10.1007/978-3-642-18638-7_6

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

[16] "PlantUML." [Online]. Available: https://plantuml.com/

[17] D. van Heesch, "Doxygen." [Online]. Available: https://www.doxygen.nl/

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

[19] W. Said, "Interactive state machine mining from embedded software," Ph.D. dissertation, University of Bremen, Germany, 2020. [Online]. Available: https://d-nb.info/1222710366

[20] T. Sen and R. Mall, "Extracting finite state representation of Java programs," *Software and Systems Modeling*, vol. 15, pp. 497–511, May 2016. [Online]. Available: https://doi.org/10.1007/s10270-014-0415-3

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

[22] M. L. Collard, M. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," in *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11)*. IEEE, 2011, p. 10 pages.

[23] J. Lesner and G. H. Elkaim, "Technical Report: State Machine Visualizer." [Online]. Available: https://github.com/jlesner/smv2/blob/main/smv_ieee_035.pdf

[24] ——, "Source Code: State Machine Visualizer." [Online]. Available: https://github.com/jlesner/smv2

[25] "Perl Regular Expressions." [Online]. Available: https://perldoc.perl.org/perlre

[26] I. Docker, "Docker." [Online]. Available: https://www.docker.com/

[27] S. Behnel and contributors, "lxml." [Online]. Available: https://github.com/lxml/lxml

[28] M. Kay and S. Limited, "libsaxonb-java." [Online]. Available: https://saxon.sourceforge.net/

[29] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.