ALLI/O Diagram: An Action-based Visual Programming Language for Embedded System

Nuntipat Narkthong*, Chattriya Jariyavajee†, Xiaolin Xu*
*Northeastern University, Boston, USA †King Mongkut's University of Technology Thonburi, Bangkok, Thailand
{narkthong.n, x.xu}@northeastern.edu chattriya.jar@mail.kmutt.ac.th

***Abstract*—This paper introduces ALLI/O Diagram, an action-based visual programming language for embedded system programming used by the ALLI/O IDE. We illustrate the practicality of ALLI/O Diagram with various design examples and evaluate it against block-based, event-based, device-based, and state-based programming approaches in terms of programming effort, readability, and portability of the result programs. These results demonstrate that our proposed ALLI/O Diagram is the most compact, expressive, and portable across different hardware models. We open source the ALLI/O Diagram and all example programs at https://allio.build.**

***Index Terms*—Embedded System, Microcontrollers, Visual Programming Language**

I. Introduction

The emerging platforms like Arduino [1] for makers and hobbyists, Mbed [2] for professional development, and graphical programming tools such as MakeCode [3] for the educational segment have significantly promoted the proliferation of embedded systems development. In terms of hardware, these platforms provide developers with a ready-to-use microcontroller (MCU) board that includes all essential circuitry needed to prototype their ideas. From a software development perspective, these platforms offer a collection of libraries that abstract away the intricate details of the hardware, which enables developers to program the MCU and all integrated peripherals with high-level functions, diminishing the learning curve and facilitating code portability across various MCUs.

One common characteristic of these platforms is that they are built on an *imperative* programming concept. However, most existing embedded systems are predominantly *reactive* systems, i.e., they must respond to multiple real-world events concurrently [4]. Although several programming techniques have been adopted to support concurrency in imperative languages, there still are many challenges. For example, the super loop pattern, although simple and efficient, can lead to spaghetti code due to the excessive use of flag variables, which is difficult to read and maintain. To address these problems, a finite state machine (FSM) or a statechart [5] can be used to systematically manage execution states. However, implementing the diagram in code requires the developer to write a significant amount of boilerplate code and manually maintain synchronization whenever the design changes. For more complex scenarios, the real-time operating system (RTOS) [2], [6]–[10] can be adopted alongside the state machine or super loop to enable real multitasking. However, it introduces steep learning curves and high resource utilization.

Alternatively, different programming strategies have been explored over the last few decades to enhance productivity and reduce the learning curve of embedded system programming. Some notable initiatives are classified as follows:

1) *Block-based programming* is the predominant method employed by educational tools. Straightforward implementations [11]–[14] replace each line of code with a graphical block and provide a ready-to-use block to program hardware devices, allowing inexperienced programmers to write code without memorizing the syntax. Nevertheless, they inherit all the shortcomings of imperative languages and may suffer in terms of readability for complex programs due to the huge view-port needed to display all blocks. More advanced implementations [3], [15]–[17] integrate some form of scheduler to facilitate event-based and concurrent execution, which helps reduce boilerplate code and improve readability for complex programs at the cost of higher resource utilization.
2) *Device-based/Dataflow programming* [18], [19] allows programmers to route values from input to output devices through some transformation nodes. Implementing systems with static behavior, such as displaying sensor values on a display or activating an actuator when sensor values exceed a certain threshold, requires minimal work. However, complex behaviors, e.g., multiple modes of operation, require building logic with flip-flops and gates, which present challenges to software developers.
3) *Trigger-action programming* [20], [21] allows developers to specify actions in response to events. Due to its simplicity and limited syntax, its use is currently limited to end-user programming tools, e.g., for connecting multiple IoT devices for home automation purposes.
4) *State-based programming* [22]–[24] uses a state diagram or statechart to describe the design and automatically converts it to code, avoiding error-prone and time-consuming manual implementation. However, most tools require developers to put code in the diagram instead of using natural language, which impacts the diagram's readability and portability.

Taking into account these design challenges and limits of the existing solutions, we believe a new programming approach that combines the strengths of existing programming methodologies is needed to improve productivity and reduce

the complexity of embedded system programming for both novice and professional developers. To that end, this paper introduces ALLI/O (pronounced as All-I-O) Diagram, a new programming approach for embedded systems that provides the following capabilities:

1) *Portable multi-purpose syntax.* ALLI/O Diagram uses high-level portable syntax that is human-readable and decouples from hardware-specific implementation detail while still being formal enough to support automatic code generation (see Sec.II-A).
2) *State-alike control-flow.* Instead of executing from top to bottom and relying on control-flow statements such as if-else, for, and while, ALLI/O Diagram allows transition to any part of the program and describes concurrent behavior with multiple diagrams similar to the statechart (see Sec.II-B3).
3) *Imperative way of thinking.* Unlike traditional state diagrams, an ALLI/O Diagram consists of a sequence of actions that modify the state of the output device and transitions that connect and trigger actions based on internal and external stimuli, hence the named *action-based* approach. Additionally, ALLI/O Diagram addresses the weakness of state-based programming in terms of program size by supporting variables, loops, and subroutines, a convenient concept from imperative programming languages (see Sec.II-B3, II-C2).

II. Design Overview

In this section, we introduce ALLI/O device abstraction, which is a portable syntax used to interface with hardware devices in ALLI/O Diagram. Then, we describe each diagram component and the overall syntax of ALLI/O Diagram using several program examples.

A. ALLI/O Device Abstraction

ALLI/O introduces the concept of *Generic Device*[1]. to represent a type of real-world device, such as an LED, a push button, a temperature sensor, etc. Each program may contain multiple Generic Devices of the same type, each identified by a unique instance name (e.g. `LED1` and `LED2` in Fig. 2). Each Generic Device consists of

1) *Action* represents a task that an output device can perform. For example, an LED can be turned on or off, a display can show some texts, a servo can move, etc. An action may require some parameters. For instance, the LED "On" action requires the brightness value and the servo "Move To" action requires the degree value.
2) *Condition* (highlighted in yellow) refers to the state of an input device that can be represented as true or false, e.g., button press, button release, accelerometer free fall, etc.
3) *Value* (highlighted in purple) represents the quantity that can be measured by the input device, e.g., acceleration, temperature, relative humidity, light intensity, etc.

[1]A complete list of Generic Devices are available at https://allio.build

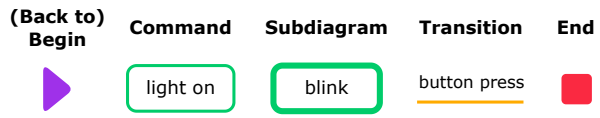

Fig. 1. The basic building block of an ALLI/O diagram.

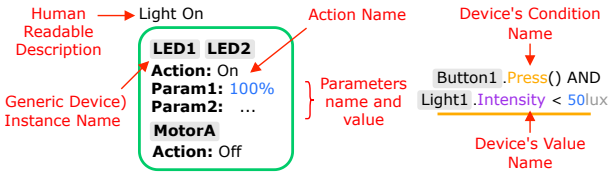

Fig. 2. Anatomy of the Command block and Transition

ALLI/O Diagram is portable across different MCUs and input/output devices as its behavior is defined based on the Generic Device. During code generation, each physical hardware device is mapped to one or more Generic Device depending on compatibility and user preferences. For instance, the Bosch BME280 sensor can measure both temperature and humidity, thus it can be mapped with two Generic Devices in the diagram. The algorithms for device mapping and code generation are left as future work.

B. ALLI/O Diagram Design

1) Diagram Component: ALLI/O diagram is a graph that begins with a `Begin` block followed by a `Command`, `Subdiagram`, or `Transition` and ends with a `Back to Begin` or an `End` block. Fig. 1 shows the design of all ALLI/O Diagram components.

`Begin` block indicate the beginning of a diagram. The `Back to Begin` block can be used to resume execution at the corresponding `Begin` block. Otherwise, the `End` block can be used to execute the diagram only once.

`Command` block is used to perform one or more actions, such as turning on a motor or displaying some text on the display. Each `Command` block contains a Generic Device instance name, along with the name of the Action and all parameters to set to the device. Each Action can be applied to multiple devices, and multiple Action/device pairs are supported in one `Command` block, e.g. in Fig. 2, `LED1` and `LED2` will be turned on and `MotorA` will be turned off.

`Subdiagram` block enables modularity and improves readability by allowing parts of the diagram to be reused and/or hidden while displaying the diagram (see Fig. 3).

`Transition` connects the output port of the source component (right-hand side) to the input port of the destination component (left-hand side) to form a graph. It is represented as a line with one boolean expression composed of one or more Conditions and/or comparison expressions joined by an AND or OR operator. A comparison expression is formed by joining two numerical expressions, each consisting of a number literal and Generic Device's Value name, with a comparison operator, e.g., `Light1.Intensity < 50` or `Temp1.Temperature > Temp2.Temperature + 10 AND Btn1.Press()`.

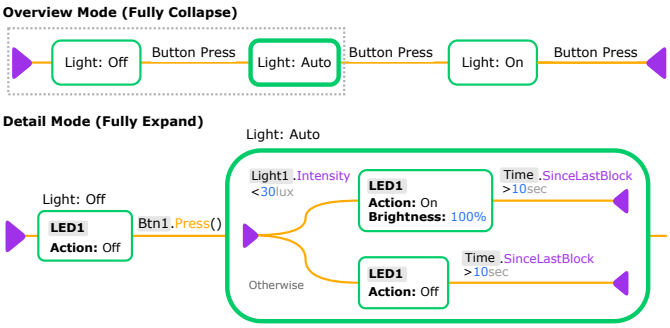

Fig. 3. A diagram describing the automatic nightlight system with three modes of operation: off, auto, and on, in two visualization modes. Only the region under the gray box is shown in detail mode due to space constraints.

2) Diagram Visualization: One common drawback of graphical programming languages is their inability to handle large designs due to the large viewport area required to visualize the program. To address this problem, ALLI/O Diagram lets users specify a human-readable description for all `Command`, `Subdiagram`, and `Transition` blocks, which will be used to render the diagram at different levels of detail, as shown in Fig. 3. In addition, the `Subdiagram` block can be collapsed to hide the implementation detail, allowing the developer to easily see the overall design in limited space.

3) Diagram Syntax: ALLI/O Diagram execution begins at a `Begin` block and proceeds from the source component to the destination component when there is no boolean expression associated with the `Transition` or when the expression is true. Note that the output port is always on the right, and the input port is always on the left. Several `Transitions` in the same level of the graph infer a branch, i.e., the execution follows the path of `Transition` that evaluates to true first. Backward connection infers a while loop, i.e., the execution repeats until the `Transition` evaluates to false. When none of the `Transitions` in the same level evaluate to true, the execution blocks until one of the `Transition` is true.

Concurrency. ALLI/O Diagram supports concurrency by allowing multiple top-level diagrams to be defined using multiple `Begin` blocks. All diagrams may refer to the same set of devices, but they must not be connected (see. Fig. 5-6).

Subdiagram. The `Subdiagram` contains one or more inner diagrams that are executed concurrently, and it can be nested. A `Subdiagram` is interruptable if it has one or more outward `Transitions` with a boolean expression. When any outward `Transition` is true, the `Subdiagram` exits, causing all inner diagrams to stop immediately, and execution resumes at the next block in the upper-level diagram. On the other hand, a noninterruptable `Subdiagram` will only exit when all inner diagrams reach their `End` block. For example, if the button in Fig. 3 is pressed during "Light: Auto" being executed, the diagram will transition to the "Light: On" block immediately. On the other hand, in Fig. 5, the "Wiper: On" block will always execute until completion as there is no boolean expression on the outward `Transition`.

Loop. One weakness of traditional state-based diagrams

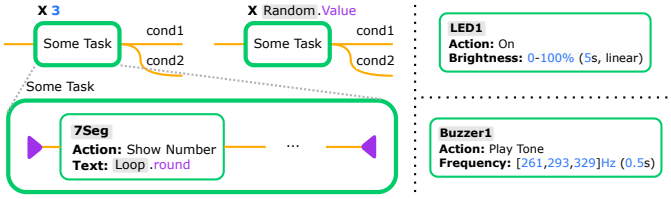

Fig. 4. A diagram syntax for repetition and animation.

is the lack of equivalent syntax to for-loop in imperative programming languages, unlike while-loop, which can be easily implemented with a backward connection. The common workaround is to unroll and create a state for each iteration of the loop, which is tedious and impacts readability. To address this problem, ALLI/O Diagram introduces additional syntax to support two common use cases of a for-loop: repetition and animation.

Fig. 4 (left) shows a syntax to support repetition by specifying the number of rounds on top of the `Subdiagram` block, which helps define the region of the repetition. When the `Back to Begin` or `End` block is encountered inside the `Subdiagram`, the loop counter increases, and the execution starts again at the beginning of the `Subdiagram`. When `cond1` or `cond2` is true, the loop exits immediately in the same way as a normal `Subdiagram`. This semantic maintains compatibility with normal `Subdiagram` and enables breaking from the loop, similar to an imperative programming language.

Animating a value is also another common use case of for-loop, e.g., dimming the light or displaying a counter on the screen. ALLI/O Diagram supports this use case by allowing the value to be specified as `start - end` in `t` seconds with an optional easing curve (see Fig. 4 (top-right)). Alternatively, a sequence of values can be provided to change the value discretely over time (Fig. 4 (bottom-right)). Note that the diagram's execution doesn't halt to wait for the value to update completely. Instead, the value will continue to update concurrently in the background. This semantic avoids blocking and allows interrupting the update by setting another Action to the device in another subsequent `Command` block.

Modularity/Parameterization. ALLI/O Diagram supports code reuse by allowing a `Subdiagram` to be reused in multiple places in the diagram where the same behavior is needed. For example, Fig. 5 shows how the "Wiper: On" block can be reused to wipe the front and/or rear windshield every 2 seconds when the corresponding switch is on.

When the same behavior needs to be applied to different sets of devices, e.g., different motor and limit switches are used for the front and rear wipers, ALLI/O allows the `Subdiagram` to be parameterized and reused by specifying the device to be used every time that it is referenced. Fig. 5 and Fig. 7 illustrate our parameterized syntax, which places the device name on top of the `Subdiagram` block. Inside the `Subdiagram`, placeholders are used, which will be mapped to the device name based on their defined order.

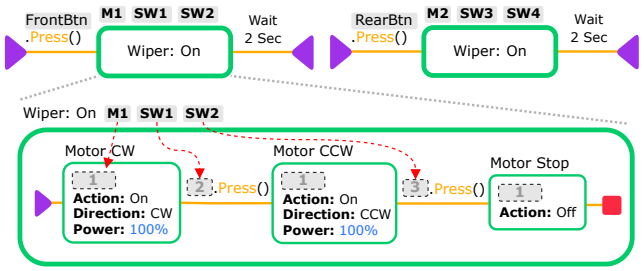

Fig. 5. A diagram describing front and rear car windshield wipers that wipe every 2 seconds when the corresponding switch is turned on.

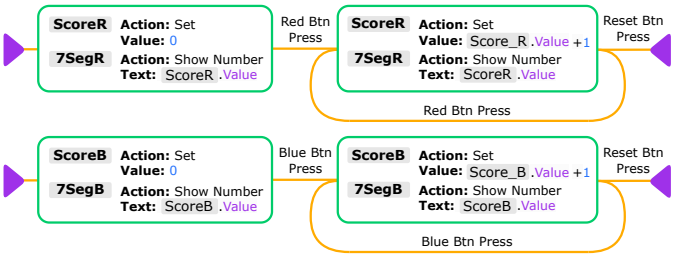

Fig. 6. A diagram describing a sport scoreboard with red and blue teams.

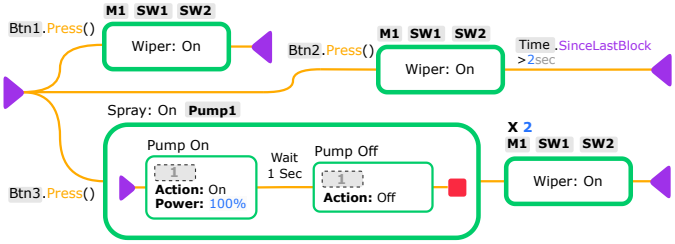

Fig. 7. A diagram describing front and rear car windshield wipers with three modes of operation: single wipe, auto wipe (every 2 seconds), and spray and wipe. Another identical diagram for the rear wiper is omitted for brevity.

C. Design Example

1) Automatic night light: Fig. 3 illustrates a complete ALLI/O Diagram to implement an automatic night light system with three mode of operation: off, auto and on. In this example, the execution begins at the `Begin` block and continues immediately to the "Light: Off" block to turn the light off. After the button (`Btn1`) is pressed for the first time, the execution enters the "Light: Auto" block to turn `LED1` on or off, depending on the intensity value read from the light sensor (`Light1`). The `Otherwise` keyword can be used to indicate a diagram path to take when all other `Transitions` are false, similar to the else clause in the if-else statement in most programming languages. After the `LED1` is turned on or off, the program waits for 10 seconds before moving on to the `Back to Begin` block, which causes the execution to resume at the latest `Begin` block. The loop continues indefinitely until the button is pressed for the second time, which causes the `Subdiagram` to exit and proceed to the "Light: On" block immediately. When the button is pressed for the third time, the diagram advances to the `Back to Begin` block and resumes at the beginning. This process repeats indefinitely.

2) Sport score board: A traditional state-based diagram requires a huge number of states to perform counting and/or memorizing values, as separate states are needed to represent each valid value. Fig. 6 demonstrates our simpler approach to implementing a score board with two buttons to increment scores for red and blue teams using two top-level diagrams and the Memory Generic Device, which offer similar functionality to a global variable in the imperative programming language. An value can be memorized with the Action "Set" and can be referred to in the diagram as a Value (e.g., `ScoreR.Value`) that can be used as an Action's parameters or in a Transition in the same way as a Value from an external sensor.

3) Car Windshield Wiper: Fig. 7 shows a diagram describing front and rear car windshield wipers with three modes of operation: single wipe, auto wipe (every 2 seconds), and spray and wipe. `Btn1` - `Btn3` represent switches in the car wiper stalk. Each wiper is modeled with 1 motor and 2 limit switch at the start and end position (see Fig. 8). The operation of the wiper is described in the "Wiper: On" block shown in Fig. 5. It works by rotating until it hits the limit switch at the end position. Then, it will rotate back in the opposite direction until it hits the limit switch at the start position and stop. This example demonstrates how to implement a branch, a loop, and a parameterized subroutine with ALLI/O Diagram.

III. Evaluations

In this section, we compare ALLI/O Diagram, our action-based approach, with the most mature tools from each existing approach, including Ardublockly [11] (imperative block programming), MakeCode [3] (event-based block programming with task scheduler), XOD [19] (device-based/dataflow programming), and SinelaboreRT [22] (state-based programming) in terms of programming effort, readability, and portability of the resulting program. We chose the single-mode front and rear car windshield wiper example in Fig. 5 as a case study because the resulting program is both small enough to fit in a limited space and complicated enough to show how each platform supports complex programs.

A. Programming Effort

Fig. 5 shows how to implement the following system in ALLI/O Diagram by simply using two top-level diagrams to describe each wiper and the parameterized syntax to reuse the `Subdiagram` between the front and rear wipers. Fig. 8 (a) shows the same program implemented in MakeCode with another forever block for the rear wiper omitted due to space constraints. Note how MakeCode allows multiple forever blocks to run concurrently, which enables developers to write simpler blocking code (e.g., loops to poll an input button and pause by 2 seconds) to implement this system. In contrast, ArduBlockly (Fig. 8 (c)), a similar block-based platform, requires the developer to manually implement a state machine with lots of if-else statements to keep track of the program's state, as the execution thread can't be blocked

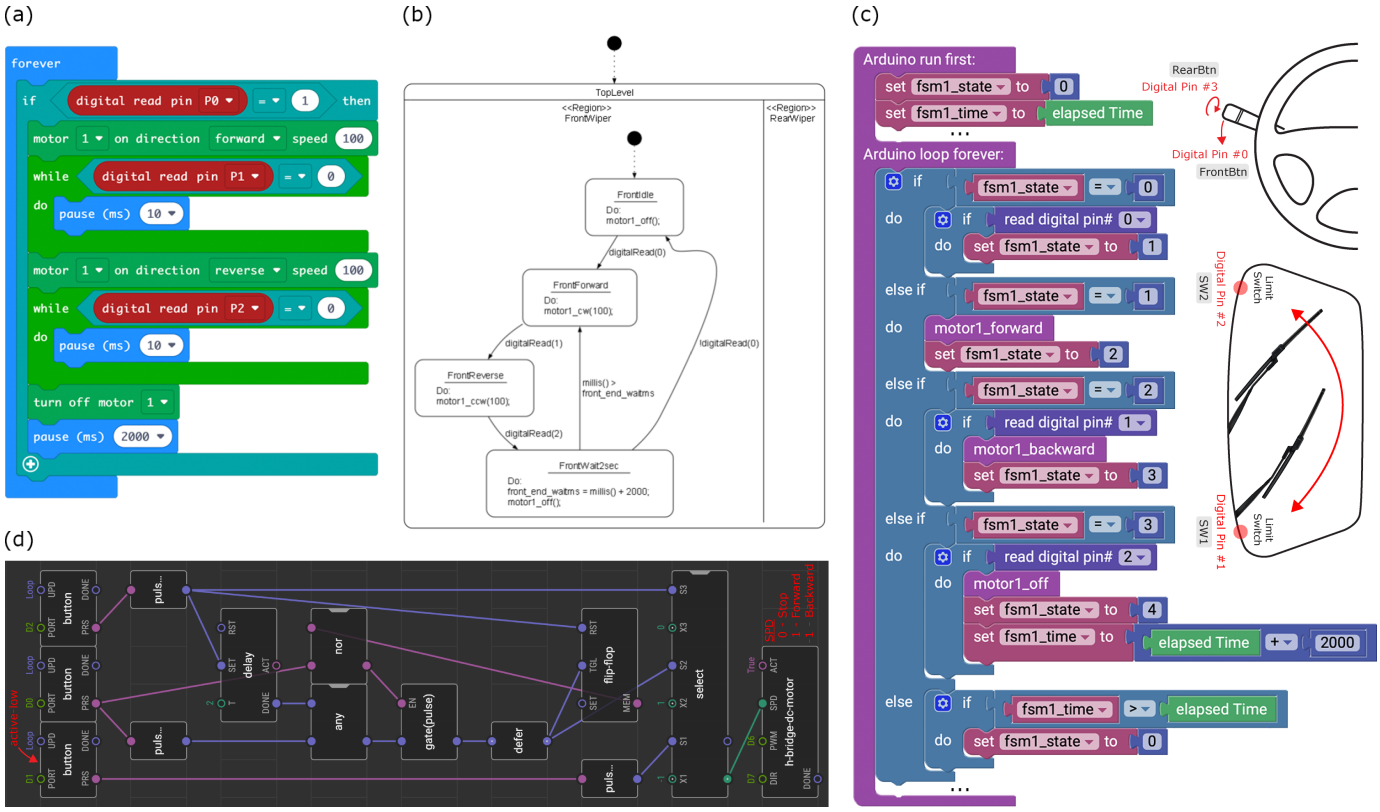

Fig. 8. Example of programs for controlling front and rear car windshield wipers on four programming platforms: (a) MakeCode, (b) SinelaboreRT, (c) Ardublockly, and (d) XOD. Only part of the program for controlling the front wiper is shown due to space constraints.

to poll a button or to wait for a specific amount of time. On the other hand, SinelaboreRT (Fig. 8 (b)) enables the developer to easily implement the same state machine logic using the statechart with code in some text-based programming languages, such as C++. Lastly, XOD (Fig. 8 (d)) requires the most effort in this case due to the need to design complex logic with the logic gate and flipflop components to manage the program state.

In summary, ALLI/O, MakeCode, and SinelaboreRT require the least effort to implement the aforementioned example, followed by Ardublockly due to its verbosity and XOD due to the use of logic unfamiliar to most software developers.

B. Readability

ALLI/O is the most readable due to its compact size and the use of human-readable Action, Condition and Value defined in the Generic Device to describe the diagram. While the SinelaboreRT uses state-based logic similar to ALLI/O Diagram, it directly puts code in the diagram, which makes the diagram large, and the reader might not be able to understand the behavior of the system without knowing the definition of each function (e.g. `motor_xxx()`). Block-based programming provides high-level block (e.g., `motor 1 on direction ...` in Fig. 8 (a)), which is user-friendly. However, it suffers when the program is more complex, e.g., in the ArduBlockly example (Fig. 8 (c)), due to the large viewport area needed to visualize the diagram. XOD is the least readable, as the complex logic may not be self-explanatory without a detailed explanation. More importantly, some crucial information, such as the initial value of a component (e.g., a flip-flop's output), is hidden from the diagram rendering.

C. Portability

ALLI/O Diagram is the only fully portable programming method, as it decouples all device-specific information (e.g., connection port, library used, etc.) from the diagram to the code generation phase. On the other hand, MakeCode, Ardublockly, and XOD programs are mostly portable across different MCU boards but not across different peripheral devices, as different devices (e.g., motor drivers) may need different blocks and interface codes. The SinelaboreRT diagram is portable across hardware devices, but the implementation of the code referred to in the diagram (e.g., `motor1_off()`) may not be portable and may need to be reimplemented by the developer before performing code generation.

IV. Conclusion and Future Works

This paper introduces the design of the ALLI/O Diagram, provides several diagram examples and compares it against other embedded system programming approaches. Our future work will investigate techniques to generate code from ALLI/O Diagram in terms of code size, latency, and power consumption on diverse MCU platforms.

ACKNOWLEDGMENT

The authors would like to thank Ingarage Assistive Technology Co., Ltd. for funding, as well as Sara Rhujitawiwat, Chaiwat Limpornchitwilai, Koetkao Sriratanaban, and Surapont Toomnark for their support and feedback on the early prototype of this work. This work was partially done while the first author was at King Mongkut's University of Technology Thonburi, Thailand.

REFERENCES

[1] M. Banzi and M. Shiloh, *Make: Getting Started with Arduino The Open Source Electronics Prototyping Platform*, 3rd ed. Sebastopol, CA, USA: Maker Media, Inc, 2014.

[2] Arm Limited. (2009) Free open source iot os and development tools from arm — mbed. Arm Limited. [Online]. Available: https://os.mbed.com

[3] J. Devine, J. Finney, P. de Halleux, M. Moskal, T. Ball, and S. Hodges, "Makecode and codal: intuitive and efficient embedded systems programming for education (lctes version)," in *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, June 2018, pp. 19–30. [Online]. Available: https://www.microsoft.com/en-us/research/publication/makecode-and-codal-intuitive-and-efficient-embedded-systems-programming-for-education/

[4] D. Harel and A. Pnueli, *On the development of reactive systems*. Berlin, Heidelberg: Springer-Verlag, 1989, p. 477–498.

[5] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167642387900359

[6] Richard Barry. (2003) Freertos - market leading rtos (real time operating system) for embedded systems with internet of things extensions. Amazon Web Services. [Online]. Available: https://www.freertos.org

[7] G. Oikonomou, S. Duquennoy, A. Elsts, J. Eriksson, Y. Tanaka, and N. Tsiftes, "The contiki-ng open source operating system for next generation iot devices," *SoftwareX*, vol. 18, p. 101089, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711022000620

[8] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, *TinyOS: An Operating System for Sensor Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 115–148. [Online]. Available: https://doi.org/10.1007/3-540-27139-2_7

[9] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "Riot os: Towards an os for the internet of things," in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2013, pp. 79–80.

[10] Zephyr Project. (2016) The zephyr project – a proven rtos ecosystem, by developers, for developers. Zephyr Project. [Online]. Available: https://www.zephyrproject.org/

[11] carlosperate. (2015) Ardublockly - embedded log. carlosperate. [Online]. Available: https://ardublockly.embeddedlog.com

[12] Sébastien Canet. (2020) Blocklyduino. [Online]. Available: https://github.com/BlocklyDuino/BlocklyDuino-v2

[13] Michael Nixon. (2016) Code kit. EduKits International Pty Ltd. [Online]. Available: https://edukits.co/code-kit-app/

[14] Keyestudio. (2021) Mixly–a superior graphical programming tool. Keyestudio. [Online]. Available: https://www.keyestudio.com/pages/mixly-a-superior-graphical-programming-tool

[15] KittenBot. (2016) Kittenblock. KittenBot. [Online]. Available: https://kblock.kittenbot.cc

[16] Makeblock. (2014) mblock block-based ide - coding for beginners. Makeblock. [Online]. Available: https://ide.mblock.cc

[17] ArtronShop. (2020) microblock ide. ArtronShop. [Online]. Available: https://ide.microblock.app

[18] Mitov Software. (2023) Visuino - visual development for arduino. Mitov Software. [Online]. Available: https://www.visuino.com

[19] XOD Inc. (2017) Xod. XOD Inc. [Online]. Available: https://xod.io

[20] IFTTT Inc. (2011) Ifttt - automate business & home. IFTTT Inc. [Online]. Available: https://ifttt.com

[21] B. Ur, M. Pak Yong Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman, "Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 3227–3231. [Online]. Available: https://doi.org/10.1145/2858036.2858556

[22] P. Mueller, "Sinelabore rt user manual," *SinelaboreRT*, 2015.

[23] Quantum Leaps, LLC. (2016) Qm model-based design tool. Quantum Leaps, LLC. [Online]. Available: https://www.state-machine.com/products/qm

[24] itemis Inc. (2022) itemis create — state machine tool - lowcode development. itemis Inc. [Online]. Available: https://www.itemis.com/en/products/itemis-create/