2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE)

Novice-Friendly Multi-Armed Robotics Programming

Nico Ritschel*, Reid Holmes*, Ronald Garcia* and David C. Shepherd†
*Department of Computer Science, University of British Columbia, Vancouver, Canada
†ABB Corporate Research, Raleigh, North Carolina, United States
Email: *{ritschel, rtholmes, rxg}@cs.ubc.ca †david.shepherd@us.abb.com

***Abstract*—Collaborative robots are being applied in a growing number of usage scenarios, but their adoption is slowed down by the high complexity of robot programming. As previous prototype studies have shown, block-based programming environments can enable novice or end users to program industrial single-armed robots. Some existing block-based tools support parallel programming and therefore show potential to be used for multi-armed robot programming as well. We analyze their designs and argue how improved abstractions and visualizations could make multi-armed parallelism accessible to novice users. Based on this analysis, we then extract a list of features that a block-based environment designed for multi-armed robot programming should provide. Finally, we present our design vision for a novel programming environment for two-armed robots, show how it provides these features and discuss how it can enable both novices and experienced intermediate users to perform parallelized programming tasks.**

***Index Terms*—Application programming interfaces, Programming environments, Robot programming, Parallel programming**

I. Introduction

Robots have advanced from highly specialized industrial machines into essential tools for a wide range of tasks. One factor that fueled this development is the rise of collaborative robots which are designed for safe use around humans [1]. As previous work has discussed, robots can be utilized in assembly lines where they interact with human workers in various ways [2]. By completing certain tasks faster and with higher precision, they can have a positive impact on both production cost and output quality.

The increasingly complex interaction between human and robot workers intensifies the demand for effective and flexible programming tools. In the past, programming environments and languages for robots were often targeted towards engineers and software developers. As such, they typically favored fine-grained control over all low-level operations performed by the robot over accessibility or intuitive usability. The resulting complexity of the tools causes them to often require years of education and training to use them effectively. Especially in smaller companies, this can slow down the adoption of robotics significantly [3].

Weintrop et. al. [4] have introduced a new programming environment called *CoBlox* for robotics that is specifically targeted towards novices with little to no programming experience. This environment is based on the concept of block-based programming, which is most commonly applied in an educational context [5]. An initial user study on CoBlox demonstrated that using a block-based environment for simple robot programming tasks significantly increased success rate, speed and satisfaction for novice users in comparison to established programming tools [4].

A comparison of studies evaluating block-based languages in education has shown that they are inherently more intuitive for novice users to learn since they prevent syntactical errors, provide additional visualization and often have more accessible user interfaces than common text-based development environments [6]. As indicated by the CoBlox user study, a block-based robotics programming environment can therefore provide an effective way to enable non-programmers to perform programming tasks that are simple but often sufficient for practical usage scenarios.

Block-based programming environments, including the CoBlox robotics programming environment, have a reduced set of language features to avoid overwhelming novice users. This can lead to the common misconception that block-based programming languages are inherently less powerful or expressive than their text-based counterparts [6]. In contrast to this belief, abstractions for complex functionalities like distributed systems [7] or specific parallel programming patterns [8] have been successfully ported to block-based languages.

One area in which block-based programming lacks maturity is enabling parallel programming. While some block-based languages offer basic functionality for multitasking, they either provide very limited abstraction, lack efficient visualization or restrict the user to a very specific set of pre-defined parallel design patterns. While some usage scenarios for block-based languages, especially in education, are inherently free of parallelism, this is typically not the case for robotics.

In this work, we particularly focus on the usage of block-based languages for programming robots with two or more arms, which show particularly high benefits in speed and efficiency when used in collaboration with human workers [9]. Simple tasks that involve interactions between the arms are the passing of objects, the simultaneous lifting of heavy loads or tool-based interactions like holding and stirring that need to be parallelized. We believe that enabling novice users to perform these tasks without advanced programming knowledge has great potential to further enable non-engineers to tailor robots to their own specific needs without outsourcing this task to expertly-trained robot programmers.

978-1-7281-2249-6/19/$31.00 ©2019 IEEE
DOI 10.1109/RoSE.2019.00013

Fig. 1. Parallelism via nested *do in order* and *do together* blocks in Alice 3.

```
public void myFirstMethod() {
    doTogether( () -> {
        this.brown_bear.move( MoveDirection.FORWARD, 0.5 );
        this.brown_bear.say( "hello" );
    }, () -> {
        this.polar_bear.move( MoveDirection.FORWARD, 1.0 );
        this.polar_bear.say( "hello" );
    } );
    this.brown_bear.turn( TurnDirection.LEFT, 1.0 );
}
```

Fig. 2. Generated Java code for the block-based program shown in Figure 1

II. Existing Approaches to Parallelism in Block-Based Programming

The type of parallelism required to control a multi-armed robot is significantly different from typical concurrent programming in software: While threads typically interact via message passing or a shared set of data, in robotics programming each parallel process shares the same fixed interface to the physical robot components. This limits the possible types of parallel interactions possible but at the same time causes a high requirement for synchronization between the arms to perform even simple tasks. This means that commonly found high-level abstractions and patterns that have been previously ported to block-based programming [8] are not applicable.

In this section, we present two previous approaches to block-based parallel programming that could potentially be applied to enable block-based robot programming for multi-armed robots. We analyze their specific strengths but also their limitations, in particular with a focus on robotics tasks.

A. Nested Sequential and Parallel Contexts in Alice 3

Block-based programming languages, while superficially different from established programming languages, typically share the same foundational features and are often even based on the same libraries. The reason for this similarity is that block-based programming is typically implemented using the same underlying syntactical structure, and often simply converted to an existing text-based language during compilation. While this means that block-based programming is not limited in its expressive power compared to classical languages, it also leads to them exhibiting the same design patterns and paradigms as textual languages [6].

The direct mapping of block-based languages to text-based patterns can be observed in parallel programming as well: Figure 1 shows a parallel program written in the block-based language Alice 3 [10] that is intended to be used in early undergraduate programming education. In the shown example, two bears in a virtual 3D scene are programmed to perform a sequence of actions: Both move forward simultaneously, then say “hello” and finally the brown bear turns left.

Alice uses two types of blocks called *do in order* and *do together* that can be nested arbitrarily to switch between a parallel and sequential context for the inserted statements. While it might not be immediately obvious how the block-based code from Figure 1 can be translated into text-based code, the generated Java code shown in Figure 2 reveals a very straight-forward mapping to a library method that instantiates separate threads for each element inside the surrounding block.

The advantage of the way Alice 3 implements parallelism is that it is intuitive to see when switches between a parallel and a sequential context happen and in particular when threads will be joined together. In the shown example, it is easy to explain that both bears will start moving simultaneously, that each bear will only say “hello” after it has moved forward, and that the brown bear will only turn left after all other commands have finished. On the other hand, the overall timeline of operations is less intuitive: An inexperienced user may not be able to immediately tell based on the code if the polar bear will “hello” before, after or simultaneously with the brown bear. Due to its structural similarity to the textual representation in Java, Alice 3’s block-based code barely provides any clarity benefits to the user.

A related issue that would affect an approach like Alice 3’s when applied to physical robots is that it may not be clear for users how contradicting commands are executed. For example, Alice 3’s underlying simulation engine can handle an object simultaneously moving forward and backward in two parallel commands by simply adding up these geometrical movements to cancel each other. In a real robot, trying to control the same motor with two different speeds may result in an error, unspecified behaviour or when executed very naively even damage to the hardware component.

B. Control Flow Visualization in Lego Mindstorms EV3

Lego Mindstorms EV3 is another example of a block-based language supporting parallelism. EV3 is a tool for programming Lego robots which is based on the LabVIEW graphical programming environment [11]. LabVIEW, targeted towards engineers and scientists simulating physical components like electrical circuits, is specifically designed with parallelism in mind: It uses a producer-consumer mechanism for managing the availability of input resources and data, and its simulation can handle even the often complex dependency structures in physical designs in parallel.

In comparison to LabVIEW, EV3 features a much simpler user interface and block design since it is more targeted towards children and other novice users with very little programming experience. However, as shown in Figure 3, it retains the basic parallelism functionality of LabVIEW: The program shown here, which is intended to be read left-to-right,

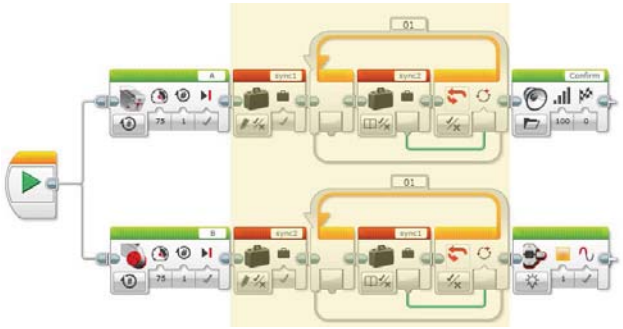

Fig. 3. Parallelism via forking in Lego Mindstorms EV3. To synchronize the threads, the user needs to manually implement a barrier (highlighted in yellow) by setting a flag in each thread and then using a loop to wait until all other threads have set theirs.

immediately splits into two separate threads (one shown on the top and one on the bottom) after it is started. In contrast to LabVIEW, threads in EV3 cannot be joined by connecting them to a single successor block but run independently. They can only be synchronized by an explicit variable- and loop-based barrier as highlighted in yellow in Figure 3, which needs to be hand-built by the user.

In comparison to Alice 3, the design of EV3 moves further away from the linearized representation of parallel operations that is typical for text-based languages. This allows it to visualize the order of execution more intuitively on a 2D grid as threads can be read as individual chains of blocks in left-to-right order. A major factor that enables EV3's more flexible layout is the free positioning of blocks and variable-length connectors like in a control-flow diagram. While this feature adds a certain amount of initial complexity, it can be particularly helpful for users familiar with the environment who can manually structure their code much more clearly by making use of EV3's full 2D canvas versus Alice 3's linear sequence of statements.

Contrasting EV3's advantages in visualization, the lack of an abstracted mechanism for joining threads is likely to pose a significant challenge to novice users. Having to write parallel barriers by hand increases the likelihood of errors that might be particularly hard to find and resolve for this target audience. Further, EV3 also lacks a mechanism that prevents or at least visualizes potential conflicts of concurrently executed commands. Manuals and tutorials for the environment explicitly warn users not to send commands to the same hardware components in multiple threads since this can cause unexpected behavior [12].

III. Our Vision of Block-Based Parallelism for Two-Armed Robots

The parallelism approaches of Alice 3 and EV3 presented in Section II provide two fundamentally different block-based representations for parallelism: One is very similar to text-based high-level languages in both the level of abstraction and linear layout it provides, while the other makes more use

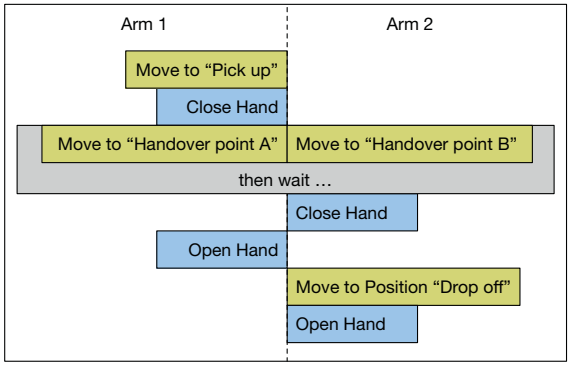

Fig. 4. Envisioned example implementation of a simple object handover task

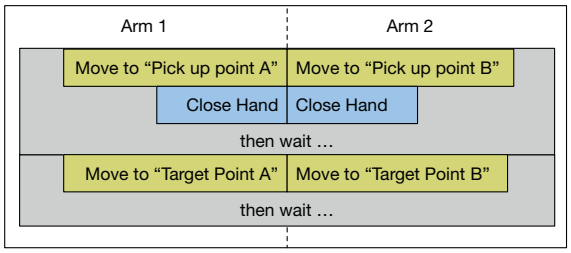

Fig. 5. Envisioned example implementation of a simple joint load carry task

of its block-based 2-dimensional visualization but provides little abstraction to its users. At the same time, both fail to provide an embedded mechanism that prevents conflicts during parallel interaction to which multi-armed robot programming is particularly sensitive.

To combine the strengths of both approaches, we envision a system that provides the following properties:

- Makes the canvas readable as a time-line, making it clear which operations are executed parallel or sequentially.
- Allows users to immediately see which actions are executed by which robot arm.
- Prevents unexpected conflicts in concurrent commands that would lead to unexpected behavior.
- Provides an abstraction for synchronizing threads that doesn't require the manual set-up of low-level constructs.

A sketch of a design targeting two-armed robots that has all of these envisioned properties can be seen in Figure 4 and Figure 5. The figures show sample programs for two usage scenarios: Handing over an object from one arm to another and using two arms to jointly carry a load. Each colored box is a block that users can drag and drop to edit the program. In addition, the canvas is divided into sections for each robot arm, and blocks are automatically associated with the arm's area they are placed in. Further, blocks are not placed freely on the canvas but are positioned in a fixed grid of time slots that could be visualized to the user by blocks "snapping" into this grid when dropped.

As only one block can be positioned per arm in each time slot, this design implicitly prevents colliding commands for the same arm. However, since we assume block heights to be uniform and not dependent on the actual duration of a single operation, there is still a need for synchronization. To achieve this synchronization, our design automatically highlights blocks that are placed concurrently (here by placing them in a grey box) and adds an explicit wait operation as a parallel barrier for both arms. In Figure 4, this synchronization is used to ensure that both arms have reached the designated handover point before continuing the process.

We intentionally decided against a potentially simpler implicit synchronization after every block since this would limit users in the ways they can program concurrent operations. In Figure 5, an explicit synchronization is required after both arms have grabbed the load before they can lift it. However, there is no need for the first arm reaching the pick-up position to wait for the other arm to arrive before it closes its gripper hand. Therefore, in our design users can decide when the final synchronization happens and add additional, potentially unsynchronized blocks to the same continuous operation.

IV. Discussion

Section III presented our envisioned design and some rather simple usage examples. In future work, we intend to build a prototype based on this design so that we can evaluate it and verify our observations in experiments with real users. There are however some open research questions relevant for actual implementations that we want to pose and briefly discuss here:

A. Handling Procedural Encapsulation

In their design for the single-threaded block-based robot programming interface CoBlox, Weintrop et. al. [4] introduce so-called robot recipes to allow users to encapsulate recurring behavior into re-usable procedures. We believe that having such a tool for structuring and creating simple abstractions is a powerful tool for users that our envisioned environment should support as well. However, procedures do not fit directly into the static grid-like timeline abstraction we envisioned.

One option to integrate procedures with our design is to move procedures into a separate canvas that provides the same layout divided by arms as our main design. A more flexible option could however be to instead allow the definition of procedures for a single arm. These could then be implicitly parametrized based on the canvas side the procedure call is placed in, applying only to this side's arm. To provide users with a maximum of design freedom, both single and multi-arm procedures could be provided. Which of these is more intuitive to use is a question that only future studies on real users can answer.

B. Designs for Different Experience Levels

Many design choices, like the ability to skip block-by-block synchronization we have presented in Section III or procedural encapsulation as described in Section IV-A, require a careful balancing of simplicity and power. A way to avoid such compromises is providing users with a choice of different interface designs for different experience levels. Alice 3 [10] that we have presented in Section II-A provides this functionality in a more limited way by disabling a few advanced features like recursion initially and requiring users to manually enable them.

An example in the context of our two-arm programming environment could be a multi-tiered design where novices initially see an interface similar to Figure 4 and Figure 5 but synchronization happening implicitly to reduce the optical clutter of the user interface. Then, when users feel like they have a solid understanding of the basic features of the environment, they can select an extended interface with all features as described in Section III. Once users have reached an even higher degree of maturity in their programming skills, more features like procedures or even multiple concurrent operations per arm could be enabled.

Acknowledgment

This work is supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

References

[1] P. J. Hinds, T. L. Roberts, and H. Jones, "Whose job is it anyway? a study of human-robot interaction in a collaborative task," *Human-Computer Interaction*, vol. 19, no. 1, pp. 151–181, 2004.

[2] J. Krüger, T. K. Lien, and A. Verl, "Cooperation of human and machines in assembly lines," *CIRP Annals-Manufacturing Technology*, vol. 58, no. 2, pp. 628–646, 2009.

[3] Z. Pan, J. Polden, N. Larkin, S. Van Duin, and J. Norrish, "Recent progress on programming methods for industrial robots," in *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*. VDE, 2010, pp. 1–8.

[4] D. Weintrop, A. Afzal, J. Salac, P. Francis, B. Li, D. C. Shepherd, and D. Franklin, "Evaluating coblox: A comparative study of robotics programming environments for adult novices," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018, p. 366.

[5] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 16, 2010.

[6] D. Weintrop and U. Wilensky, "To block or not to block, that is the question: students' perceptions of blocks-based programming," in *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM, 2015, pp. 199–208.

[7] B. Broll, A. Lédeczi, P. Volgyesi, J. Sallai, M. Maroti, A. Carrillo, S. L. Weeden-Wright, C. Vanags, J. D. Swartz, and M. Lu, "A visual programming environment for learning distributed programming," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 2017, pp. 81–86.

[8] A. Feng, E. Tilevich, and W.-c. Feng, "Block-based programming abstractions for explicit parallel computing," in *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*. IEEE, 2015, pp. 71–75.

[9] S. Kock, T. Vittor, B. Matthias, H. Jerregard, M. Källman, I. Lundberg, R. Mellander, and M. Hedelind, "Robot concept for scalable, flexible assembly automation: A technology study on a harmless dual-armed robot," in *Assembly and Manufacturing (ISAM), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 1–5.

[10] S. Cooper, W. Dann, and R. Pausch, "Alice: a 3-d tool for introductory programming concepts," in *Journal of Computing Sciences in Colleges*, vol. 15, no. 5. Consortium for Computing Sciences in Colleges, 2000, pp. 107–116.

[11] R. Jamal, "Graphical object-oriented programming with labview," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 352, no. 1-2, pp. 438–441, 1994.

[12] D. Benedetelli, *LEGO MINDSTORMS EV3 Laboratory: Build, Program, and Experiment with Five Wicked Cool Robots*. No Starch Press, 2013.