CI/CD Meets Block-Based Languages

Hugo da Gião*, Rui Pereira†, Jácome Cunha*
hugo.a.giao@inesctec.pt, rui.alexandre.pereira@outsystems.com, jacome@fe.up.pt
*Faculty of Engineering, University of Porto & HASLab/INESC TEC, Portugal †OutSystems, Portugal

***Abstract*—Continuous Integration and Continuous Deployment (CI/CD) pipelines play a vital role in the DevOps process, enabling developers to automate and enhance software delivery. However, the existence of multiple technologies, such as GitHub Actions, GitLab CI/CD, or Jenkins, poses challenges due to their lack of interoperability and the use of different programming languages for pipeline construction.**

To address these challenges and improve the CI/CD process, our objective is to develop a block-based language specifically designed for representing CI/CD pipelines. With our language, we intend to empower users to more easily create correct pipelines. Through an interactive and user-friendly process, our approach guides users in constructing pipelines, ensuring accuracy and reducing errors. Additionally, our language will facilitate seamless transitions between different pipeline technologies, providing users with flexibility and ease of adoption.

***Index Terms*—DevOps, CI/CD, Block-Based Languages, Blockly, Visual Languages**

I. Introduction

In the realm of modern software development, CI/CD has emerged as a means for organizations to achieve rapid and frequent delivery of changes. A CI/CD pipeline encompasses a series of essential steps involved in integrating and deploying codebase changes [1]. This process involves the regular integration of new code changes, which undergo automated building and testing procedures. Subsequently, the validated code is deployed into production through the CD process [1].

Traditionally, organizations have had the option to employ various technologies like GitHub Actions[1], GitLab CI/CD[2], or Jenkins[3] to accomplish these tasks. However, these technologies often employ non-interoperable languages, come with distinct learning curves, and lack the flexibility to seamlessly switch between underlying technologies [1].

In light of these challenges, our objective is to harness the potential of block-based languages to streamline the creation of CI/CD pipelines. By adopting our approach, users can define their pipelines in a visual and interactive environment that proactively detects errors and offers helpful suggestions. Through block-based programming, we can also enforce pipeline construction rules, reducing the likelihood of constructing invalid pipelines compared to using languages such as YAML. Additionally, our method enables users to effortlessly switch between different providers without requiring pipeline rewrites. This is possible since, from the block-based program, we can generate different target languages.

In this paper, we describe our vision which, upon completion, we intend to empirically validate with CI/CD experts to assess the gains in efficiency and efficacy.

II. Related work

Buddy[4] is a comercial visual platform designed for the creation and execution of CI/CD pipelines. It offers several advantages, including increased adoption of CI/CD practices and several optimization features such as caching and parallelism. Additionally, Buddy provides seamless integration with leading development tools and platforms like Amazon AWS[5] and Microsoft Azure[6]. Our approach distinguishes itself by leveraging block-based languages for pipeline construction. Moreover, the pipelines created using our approach can be ported to various existing platforms. Furthermore, our approach enhances the development experience by assisting users in identifying and rectifying errors during pipeline construction, a feature not available in Buddy's interface.

Tegeler et al. [2] present their innovative approach to graphically modeling CI/CD workflows. The authors propose a model-driven strategy to tackle the challenges associated with maintaining complex CI/CD workflows and introduce the concept of graphical modeling using Rig. To validate their approach, the authors construct a typical web applications pipeline. In contrast, our objective is to develop a block-based language, which has demonstrated success in other contexts. Additionally, we plan to conduct extensive validation using a wide range of use cases and incorporate features that aid users in constructing accurate pipelines. Furthermore, our goal involves validating the tool through user feedback and input.

Similarly, Piedade et al. [3] propose a low-code approach for container orchestration. Their approach utilizes a visual notation that allows users to create Docker[7] files. By supporting all elements of the orchestration file, their approach enhances usability, reduces errors, and accelerates development time. The authors validate their approach through a user study involving novice developers. Similarly, we aim to design a language capable of generating scripts for existing tools, albeit with a block-based paradigm instead of a new language.

The first author was supported by the doctoral grant 2022.13084.BD financed by the Portuguese Foundation for Science and Technology (FCT).

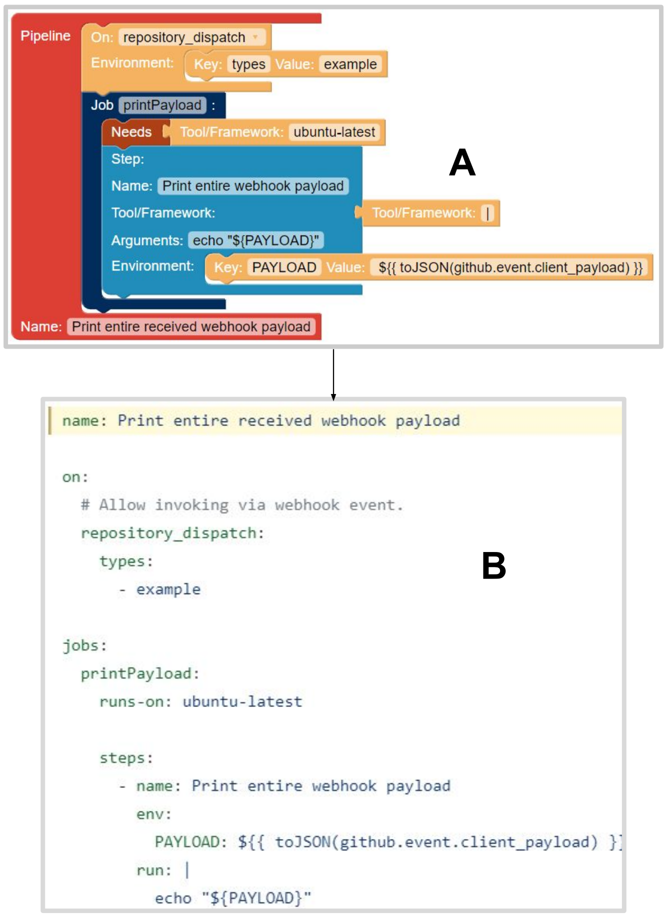
The third author was financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDP/50014/2020.

[1]https://github.com/features/actions
[2]https://docs.gitlab.com/ee/ci/
[3]https://www.jenkins.io/
[4]https://buddy.works
[5]https://aws.amazon.com
[6]https://azure.microsoft.com
[7]https://www.docker.com/

Furthermore, our focus in on different aspects of the DevOps process, offering unique functionalities and capabilities.

III. Block-Based CI/CD

In this section, we present an illustrative example that demonstrates our envisioned block-based language, designed to enable users to express their CI/CD pipelines with efficiency and clarity. Our language comprises a diverse set of blocks defining the components constituting a pipeline. These components encompass instructions for executing, defining environment variables, and specifying the jobs, steps, and tools required for each job within the pipeline.



```
Pipeline
  On: repository_dispatch
  Environment: Key: types Value: example
  Job printPayload :
    Needs Tool/Framework: ubuntu-latest
    Step:
    Name: Print entire webhook payload
    Tool/Framework: Tool/Framework: |
    Arguments: echo "${PAYLOAD}"
    Environment: Key: PAYLOAD Value: ${{ toJSON(github.event.client_payload) }}
Name: Print entire received webhook payload
```

```yaml
name: Print entire received webhook payload

on:
  # Allow invoking via webhook event.
  repository_dispatch:
    types:
      - example

jobs:
  printPayload:
    runs-on: ubuntu-latest

    steps:
      - name: Print entire webhook payload
        env:
          PAYLOAD: ${{ toJSON(github.event.client_payload) }}
        run: |
          echo "${PAYLOAD}"
```

Fig. 1. Representation using our block-based language (top part of the image) of a GitHub Actions pipeline (bottom).

In Figure 1, we present an example of a pipeline written using GitHub Actions[8] (B) and the corresponding block-based representation using our proposal (A). The depicted pipeline represents a GitHub Actions pipeline designed to print the webhook payload to the console.

To construct the pipeline using our language, we employ a $Pipeline$ block. This block takes arguments for the pipeline's name (as a string) and the content of the pipeline (as a list of blocks). The available blocks include Job, On, $Environment$, and $Needs$.

To specify the pipeline's execution trigger, we incorporate an On block and select the option *repository_dispatch*. This indicates that the pipeline should be triggered when a repository dispatch event occurs. We also include an $Environment$ block within the block to define the specific type of dispatch that activates the pipeline. An environment allows the user to define key-value pairs that can be used to record variables and their values, which can then be accessed within the context where they are defined.

After defining the pipeline's trigger, we add a job to the pipeline responsible for printing the webhook content. To accomplish this, we include a Job block within the pipeline's block list. We assign a name to this job and then provide a list of blocks representing the necessary tool and framework requirements for the job and its subsequent steps. Firstly, we include a $Needs$ block within the job's content. This block accepts a $Tool/Framework$ block, indicating that this job requires the latest version of Ubuntu. Next, we add a $Step$ block, which encompasses the $Name$ of the step, a $Tool/Framework$ block (denoted as $|$), representing a shell script in GitHub Actions, and the arguments comprising the contents of the shell script. Finally, we incorporate an $Environment$ block to store the webhook content under the key *Payload*. Within the step's arguments, we access the contents of the webhook by referring to this key.

The names we use for the blocks are the same used by the GitHub Actions syntax, as this is one of the most used languages for CI/CD (since it is part of GitHub). However, we intend to generate CI/CD code in other languages, such as Jenkins or GitLab's.

IV. Conclusions and Future Work

This paper presents a vision for a block-based approach for defining CI/CD pipelines, introducing a set of blocks that constitute the language. We demonstrate the practicality of these blocks through a real-world example.

Moving forward, our objectives involve properly defining all the constructs and their semantics, including defining how the blocks can be composed together.

Additionally, we aim to integrate the language into a visual environment tailored for CI/CD pipeline development. In this environment, users will utilize the block-based language to construct their pipelines, while our environment compiles and translates the pipeline in real-time to their desired programming language or technology, such as GitHub Actions, GitLab CI/CD, or Jenkins. By imposing restrictions on how blocks can be assembled and providing informative warnings and error messages, our approach enhances pipeline correctness and assists users in rectifying mistakes.

To evaluate the usability and effectiveness of our visual environment, we will conduct a comprehensive usability study.

[8]The pipeline example pertains to a GitHub API and GraphQL client application and was taken from https://github.com/MPLew-is/github-api-client/blob/cd20f744ff0e7789896cb745b90f965821898d41/Examples/GithubActionsWebhookClient/print-payload.yaml#L4

References

[1] G. Kim, J. Humble, P. Debois, and J. Willis, *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*, ser. ITpro collection. IT Revolution Press, 2016. [Online]. Available: https://books.google.pt/books?id=ui8hDgAAQBAJ

[2] T. Tegeler, S. Teumert, J. Schürmann, A. Bainczyk, D. Busch, and B. Steffen, "An introduction to graphical modeling of ci/cd workflows with rig," in *Leveraging Applications of Formal Methods, Verification and Validation*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2021, pp. 3–17.

[3] B. Piedade, J. P. Dias, and F. F. Correia, "Visual notations in container orchestrations: an empirical study with docker compose," *Software and Systems Modeling*, vol. 21, no. 5, pp. 1983–2005, Oct 2022. [Online]. Available: https://doi.org/10.1007/s10270-022-01027-8