Bringing a continuous integration and delivery pipeline into MoVES

Marcus Mörtstrand, Alessio Bucaioni†, Federico Ciccozzi†
† Mälardalen University (MDU), Västerås, Sweden; name.surname@mdu.se

***Abstract*—The increasing complexity of modern automotive software systems has prompted the development and adoption of various model-based methodologies. One such methodology is MoVES, which provides automated support for identifying design solutions that meet timing requirements, thus enhancing the cost-efficiency of automotive software development.**

To further enhance the cost-efficiency of the automotive software development process, we propose the integration of a continuous integration and delivery pipeline into MoVES. In addition to reduce manual tasks, this extension offers several improvements over traditional continuous integration and delivery pipelines with respect to the performance, parallel execution and distribution of the pipeline.

We begin by discussing the overarching design of the pipeline, followed by its practical implementation using GitHub Actions. Subsequently, we validate the effectiveness of our pipeline through a use case involving brake-by-wire and wiper motor functionalities in automotive systems. Finally, we describe the research challenges encountered during this process and provide an outline of future research aimed at further enhancing the pipeline.

***Index Terms*—Model-based development, automotive software, DevOps.**

I. Introduction

Over the past few decades, the automotive industry has experienced a significant rise in the complexity of embedded automotive software. Nowadays, vehicles can be equipped with over 150 million lines of code [1]. To tackle this growing complexity and facilitate the development of embedded software, the automotive industry has embraced various advancements in software engineering, including model-driven engineering (MDE) [2]. MDE enables the automatic transition from abstract models [3] to more concrete representations, eventually leading to code generation [4]. Model transformations play a crucial role in automating the manipulation of these models [5]. Several studies have explored the use of models and model transformations to create model-based development methodologies and frameworks [6]–[8]. However, most existing model-based methodologies exhibit poor integration with software development life cycle processes [9]. This limitation becomes particularly significant when adopting agile processes like DevOps [10], [11], as the lack of full-fledged integration can undermine the advantages offered by both model-based methodologies and agile practices. Consequently, the development of embedded automotive software may incorporate manual steps due to the absence of seamless integration.

In this research, we design and implement a continuous integration and delivery (CI/CD) pipeline to bridge the gap between model-based methodologies and DevOps processes. A CI/CD pipeline automates various manual interventions typically involved in transitioning new code from a commit into production, spanning the build, test, and deploy phases. We build upon the MoVES methodology, which is a model-based approach that facilitates the development and architectural exploration of automotive system designs with temporal awareness [12]. Following the engineering method outlined by Basil [13], we extend MoVES by integrating a CI/CD pipeline. This extension incorporates parallel execution of jobs and the establishment of a distributed pipeline, with each job serving as a system node. Hence, besides reducing the need of manual tasks, it provides several improvements over traditional CI/CD pipelines with respect to performance and compatibility requirements. In addition, we validate the introduced CI/CD pipeline by means of two use cases mimicking the brake-by-wire and the wiper motor automotive functionalities. Eventually, we discuss the research challenges encountered during the design and development of the pipeline and outline future directions for further enhancements.

The remainder of this work is as follows. Section II provides the background information for this research. Section III outlines the design and implementation of the pipeline. Section IV presents the application of the pipeline to the brake-by-wire and wiper motor functionalities in the automotive domain. Section V explores alternative approaches to the proposed pipeline implementation. Section VI discusses relevant work conducted in the field. Section VII concludes the paper, offering final remarks and suggestions for future research endeavours.

II. Background and motivation

In this section, we provide an overview of MoVES and describe the gasps motivating this research. In addition, we delve into CI/CD practices and the technologies used for the implementation of the proposed pipeline. In a

A. MoVES

MoVES is a model-based methodology for the development and architectural exploration of automotive software designs with temporal awareness that are identified using analytical as well as trace-based timing analysis [12], [14]. Figure 1 provides a simplified graphical representation of MoVES[1]. MoVES uses the EAST-ADL automotive modelling language

[1]It is worth remarking that the comprehensive description of MoVES it is outside the scope of this work. The interested reader may refer to the following publications [12], [14]

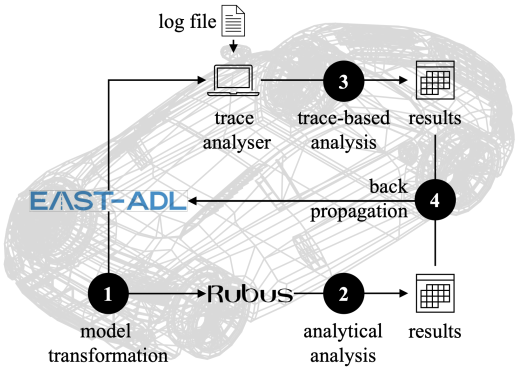


Fig. 1: Simplified representation of MoVES.

to represent software functionalities [15]. The EAST-ADL models comprehensively capture various aspects of automotive functionalities, including functional aspects, timing, verification, and validation concerns. The timing information also encompasses the timing constraints that require verification. To achieve this, two distinct processes are employed, as illustrated in Figure 1. In the first process, a set of model transformations automatically generate Rubus Component Model (RCM) models from the EAST-ADL models [12]. These RCM models are verified using pre run-time, high-precision schedulability analyses[2] available through the Rubus Integrated Component Model Development Environment [15] (Rubus ICE). The results obtained from the schedulability analyses are automatically back-annotated to the EAST-ADL models using an additional set of model transformations [12]. In the second process, the EAST-ADL models are combined with a log file that captures the simulation or actual execution of the modelled functionality [14]. The log file contains information such as the values of the modelled signals and corresponding timestamps. The models and the log file are fed to the trace analyser that analyses the traced data and generates results that are subsequently back-annotated to the EAST-ADL model, as described earlier [14].

Despite its capability to support the automated exploration of temporal-aware automotive software designs, MoVES lacks seamless integration with agile processes like DevOps, which involve frequent transitions of new artefacts from commits to, e.g., build and deployment phases. As a result, the verification processes required by MoVES must be manually executed each time the EAST-ADL model or the log file undergo modification. To address this limitation, we design and implement a CI/CD pipeline and include the pipeline in MoVES.

B. DevOps and CI/CD

DevOps is an agile software process that aims to bridge the gap between software development (dev) and operations (ops). While there is no universally standardised definition of DevOps, it is widely recognised as a methodology that strives to minimise the time between making a system change and deploying it into production [17]. In recent years, there has been a significant increase in the number of companies adopting DevOps. According to a recent report [18], approximately 61% of the surveyed companies claim to utilise this approach. The DevOps life-cycle involves a range of activities that are executed iteratively and repetitively. These activities encompass code development, planning, monitoring, operations, deployment, release management, testing, and building. While CI/CD shares the same goal as DevOps, it can be viewed as a specific tactic within the broader DevOps process. CI/CD is characterised by a set of development practices that enable the seamless integration and continuous delivery of software. A CI/CD pipeline comprises a series of steps that must be executed to successfully deliver a new version of the software. Typical steps in a general CI/CS pipeline are commit change, trigger build, build, notify of build outcome, run tests, notify of tests outcome, deliver build to staging, deploy to production.

III. The Pipeline

In this section, we discuss the design and implementation of the proposed pipeline. The interested reader can access the full implementation at https://github.com/MarcusM94/dev-env

A. Designing the pipeline

During the development of the CI/CD pipeline, we had to consider several key aspects to ensure its compatibility with best practices while enable continuous integration and continuous deployment to MoVES.

The first aspect we addressed was the handling of test failures. In a conventional CI/CD pipeline, the failure of a test would halt the entire pipeline and notify the developers of the failure. However, this approach could not be directly applied to MoVES. In fact, within MoVES it is essential to retain the timing analyses outputs and store them back into the EAST-ADL model. As a result, the only stages in the pipeline that could lead to its failure are the building stages of the two timing analysis processes. An example of a build error entails encountering an incorrect file path or having a dependency on outdated software. However, our objective was to avoid deploying a modified EAST-ADL model that would fail specific timing requirements. To achieve this, we decided to consistently deploy the modified EAST-ADL model to a development branch. Nevertheless, we proceed with deploying it to a production branch only if none of the timing analyses produce a failure.

Another aspect we addressed was the parallel execution of the pipeline's jobs. In a typical CI/CD pipeline, the tests, also referred to as jobs, are executed sequentially. However, this approach has certain drawbacks. Firstly, if the first job fails to build, the second job will not be executed, resulting in a failure to notify the developer. In the context of MoVES case, this is sub-optimal as MoVES involves two distinct timing analysis processes. Considering this, we decided to build a pipeline capable of parallel execution.

[2]The interested reader may refer to the work by Mubeen et al. for further information on the supported schedulability analyses [16]

The last aspect that we addressed pertains the distribution of the pipeline. MoVES uses Rubus ICE that in turns requires a Windows machine. However, the use of a Windows machine imposes certain limitations. For instance, certain software can not be installed via a command-line interface. Many software tools with Windows dependencies mandate installation through an installation wizard, thereby requiring user interaction with a graphical interface. Even the installation of Git on a Windows machine can prove cumbersome without graphical user interaction. If one aims to leverage the benefits of Docker in adopting a more DevOps-oriented approach to development, the installation of software must occur within a text-based interface, as specified in a dockerfile. To solve this drawback, we decided to construct the pipeline as a distributed system, employing distinct nodes, each capable of running various operating systems or different versions of a specific operating system.

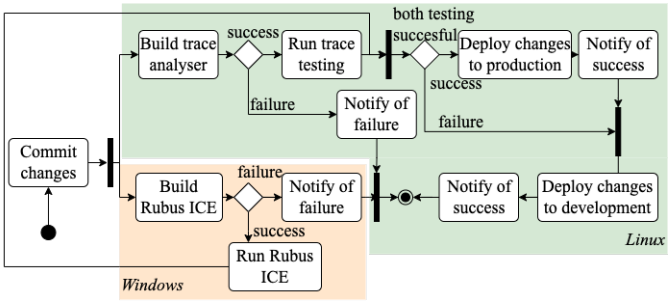


Fig. 2: Proposed pipeline

Considering the aforementioned aspects, Figure 2 illustrates a UML activity diagram that outlines the sequential steps of the proposed pipeline, which are executed whenever updates are made to the initial EAST-ADL model. We developed this pipeline using two nodes: one designated for a Linux machine (green in Figure 2) and another for a Windows machine (yellow in Figure 2). The latter node is responsible for the Rubus ICE build, which ensures analytical timing analysis. The former node handles the build of the trace analyser, guaranteeing trace-based analysis. If either of the builds encounters an error, the pipeline notifies the failure and halts further execution. Conversely, if the builds succeed, the pipeline proceeds with both the analytical and trace-based timing analyses, as described in Section II. Upon obtaining positive results from these analyses, the pipeline proceeds to the deployment to the production and development branches. However, if any of the analyses yield unsatisfactory results, the pipeline restricts the deployment only to the development branch.

B. Implementing the pipeline using GitHub Actions

We implemented the pipeline using GitHub Actions. To this end, we created a dedicated directory within our repository named "`workflows`" that contains the YAML file. This file defines the GitHub Actions pipeline and incorporates all the necessary scripting for individual job execution. GitHub Actions encompasses a feature known as "`runners`", which essentially comprises operating system images. These images typically come equipped with pre-installed software and determine the specific operating system and software utilised for each job [19]. It is worth noticing that while runners eliminate the need for Docker images, developers still retain the option to employ Docker images if required.

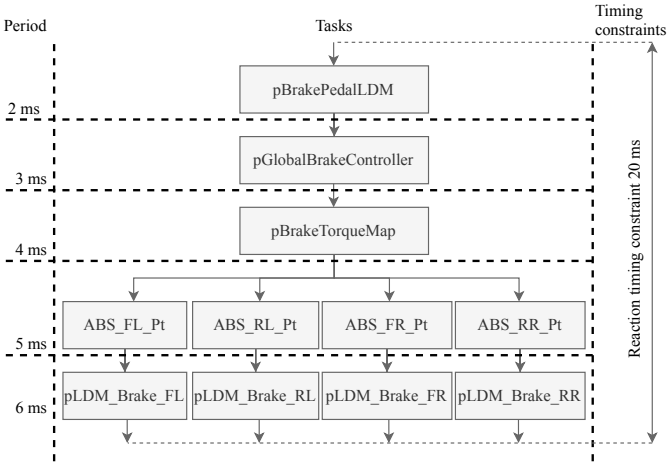
```
12  jobs:
13    trace-test:
14
15      runs-on: ubuntu-latest
16      strategy:
17        fail-fast: false
18        matrix:
19          python-version: ["3.8", "3.9", "3.10"]
```

Fig. 3: YAML file and example of a runner

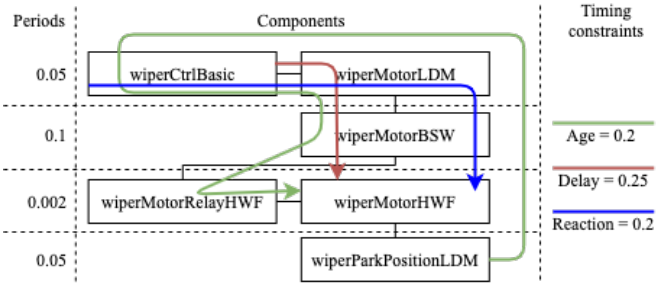
Figure 3 exemplifies a job named "`trace-test`" utilising a runner named "`ubuntu-latest`". By specifying this runner, we ensure that the job executes on the most up-to-date version of the Ubuntu operating system. To overcome the challenge posed by the need for graphical user interaction in Rubus ICE, we leveraged a GitHub Actions feature known as "`self-hosted runners`" This feature allowed us to manually install Rubus ICE on a Windows machine and integrate it into our existing pipeline as a node. After manually installing Rubus ICE using a self-hosted runner, we were able to utilise its functionality through a text-based interface, eliminating the need for graphical user interaction.

IV. Applying the Pipeline to Automotive Functionalities

This section highlights the utilisation of our proposed pipeline on two automotive functionalities: brake-by-wire (BBW) and wiper motor (WM). The BBW system revolutionises traditional mechanical linkages by introducing an independent braking system that enables electronic control of the brakes. The WM system is responsible for activating the mechanisms that operate the windshield wipers in a vehicle. It is worth noting that there exist various implementations of these systems. However, for this study, we have collaborated with an internationally renowned Swedish automotive Original Equipment Manufacturer (OEM). Hence, we have utilised the specific implementation provided by our OEM partner. Figure 4 visually depicts the configuration of the BBW and WW systems. The BBW system consists of 11 periodic tasks, each activated at five distinct periods. These tasks are responsible for executing the necessary actions whenever the brake pedal is pressed, ensuring prompt and precise activation of the brakes within a defined time frame. To achieve this, the architecture specifies four data chains, each subject to specific reaction timing constraints. The first three tasks (`pBrakePedalLDM`, `pGlobalBrakeController`, and `pBrakeTorqueMap`) are shared among all data chains. From there, each data chain continues independently for each wheel, e.g., `ABS_FL_Pt` and `pLDM_Brake_FL` for the front left wheel. Similarly, the WM system consists of six periodic tasks activated at three distinct periods. To achieve the correct functionality, the architecture specified three data chains (marked with green,



(a) Periodic activation and timing constraints of the BW system.



(b) Periodic activation and timing constraints of the WM system.

Fig. 4: Block diagram of the BBW and WW systems. Different tasks and their communication pattern are shown. Task period and reaction time constraints are annotated.

blue and red lines in Figure 4b) subject to age, delay and reaction constraints. Below we present the successful execution of the proposed pipeline on the two systems using the MoVES methodology. We use the BBW system to validate the Rubus ICE testing and the WW system for the trace analyser. To initiate the pipeline, the EAXML file of the BBW and WM systems were uploaded to a designated GitHub repository. Figure 5 showcases the interface of GitHub Actions, where all the triggered jobs are displayed upon any changes made to the repository. In our pipeline setup, we explicitly define that a push event to the Git repository will activate the pipeline. The jobs within the pipeline are responsible for both executing the tests and configuring the testing environment.

In the proposed pipeline, the Rubus ICE job operates differently from the other jobs, as it runs on a self-hosted runner hosted on a private Windows machine instead of GitHub Actions' own cloud. This dedicated Windows machine already has Rubus ICE pre-installed, enabling the utilization of a text-based interface for testing purposes. To ensure the proper functioning of Rubus ICE, the installation of Visual C++ Redistributable is required, as it installs the necessary C and C++ run-time libraries. During the execution of the Rubus ICE job within this pipeline, two essential steps are performed. Firstly, a tool called Rubus Compiler, which is a component

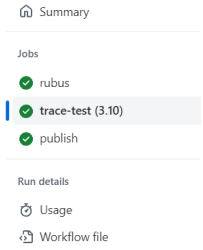


Fig. 5: Interface of GitHub Actions.

of the Rubus ICE tool suite, is utilised. This tool plays a crucial role in creating an intermediate representation of an XML file. It verifies the correctness of the XML syntax and ensures that all entries adhere to C code naming conventions. The result of this compilation process is a binary file that will serve as the input for testing purposes. The second and final step in this job involves using the Rubus Test tool to conduct testing on the previously compiled file. This tool enables comprehensive testing of the binary file, ensuring the reliability and functionality of the system.

Unlike the Rubus ICE job, the trace analysis job in the pipeline does not require the use of a Windows machine. Therefore we can leverage GitHub Actions runners available in their cloud infrastructure to perform the necessary testing. This allows us to fully embrace DevOps practices and configure the testing environment through a text-based interface. To set up the environment for the trace analysis job, the following steps are executed. The job begins by downloading the trace analyser from its repository. This ensures that the latest version of the trace analyser is obtained for testing purposes. Next, the job downloads and installs .NET Core (specifically version 3.1 as it corresponds to the one used during the development of the trace analyser). This step ensures that the appropriate version of .NET Core is available for building and running the trace analyser. With .NET Core successfully installed, the job proceeds to build the trace analyser using the framework. This process involves compiling the project and its dependencies into a set of binaries. These binaries form the executable that will be used to run the trace analyser. Finally, the job conducts the necessary testing using the built trace analyser. This step ensures the thorough examination and analysis of the trace data, enabling the identification of any potential issues or anomalies.

Figure 6 visually represents the output obtained from the trace testing job within our pipeline. The trace analysis is conducted successfully and its results highlight that the current design of the WW system does not meet the specified timing requirements. Specifically, the delay constraint of 0.25 milliseconds and the reaction constraint of 0.2 milliseconds are not satisfied. This outcome is consistent with the explanation provided in Section II, where it was clarified that the failure to meet these timing requirements does not indicate a pipeline failure. Rather, it indicates that the current design does not

adhere to the specified timing constraints. As per the MoVES methodology, the results obtained from the trace analysis are propagated back to the EAXML file. This iterative process allows for refinement and improvement of the system's design, incorporating the findings from the trace analysis to optimise the timing behaviour of the WW system.

V. Discussion

In this study, we have successfully integrated a CI/CD pipeline into the MoVES methodology. This integration involved multiple iterations of the engineering method, extensive research, and thorough testing of various products and tools that aid in streamlining the creation and management of CI/CD pipelines.

Initially, we attempted to create the pipeline using Jenkins, an open-source product that offers numerous advantages over other alternatives. While Jenkins emerged as the most beneficial choice due to its open-source license, we encountered a limitation during our work that necessitated the use of an alternative product to complete the automation process. Specifically, Rubus ICE required a Windows machine, which posed challenges in configuring Jenkins to accommodate this requirement. As our research goal focused on creating a CI/CD pipeline aligned with the DevOps methodology, attempting to tailor Jenkins to our needs gradually deviated from DevOps best practices. Notably, containerising each job in the pipeline and fully utilising a text-based interface for configuration became challenging on a Windows machine.

Additionally, we desired to host our pipeline in the cloud, enabling its utilisation across multiple machines. However, when attempting to migrate the Jenkins pipeline to cloud services like AWS, Azure, and Google, we encountered another limitation. The free-tier cloud services offered by these providers, such as AWS EC2, were insufficient for hosting the Jenkins pipeline due to limited resources. For instance, the AWS free-tier EC2 instance only provided 1GB of RAM, with a portion already allocated to the operating system [20]. We deemed a minimum of 2GB of dedicated RAM necessary for the Jenkins pipeline. Consequently, this further motivated us to switch to an alternative CI/CD pipeline creation tool.

Various approaches can be adopted to fulfil our research goal, and our work has resulted in the creation of a CI/CD pipeline with several notable improvements compared to conventional pipelines. One significant enhancement is the parallel execution of all jobs, as well as the distribution of the pipeline across multiple nodes. Although we implemented our final pipeline using GitHub Actions, it is important to note that other products such as Jenkins, GitHub Actions, and GitLab can also support parallel execution and the creation of distributed pipelines. These products are designed with similar principles since CI/CD pipelines are a fundamental part of the DevOps methodology, adhering to specific standards. It is worth highlighting that researchers and practitioners are not limited to using GitHub Actions alone if they wish to replicate our pipeline. The options we researched, such as Jenkins, GitHub Actions, and GitLab, provide comparable functionalities in terms of parallel execution and distributed pipeline creation. Ultimately, the choice of tool depends on individual project needs, time constraints, and the level of customisation desired.

VI. Related work

The adoption of DevOps processes has experienced a significant increase in software development, especially in the context of CI/CD pipelines. In this section, we discuss the importance of various studies conducted in the domains of MDE, automotive software, and CI/CD, highlighting their relevance to our own research.

Lwakatare et al. conducted a comprehensive study on the challenges of implementing DevOps practices in the embedded systems domain, including the automotive industry [21]. Lwakatare et al. found that DevOps, being predominantly associated with System as a Service (SaaS) applications, lacks considerations that make it directly applicable to the embedded systems domain, specifically within the automotive industry. Their study involved an in-depth analysis of four different enterprises including an automotive company. They found that testing embedded systems posed significant difficulties due to the intricate nature of these systems that involve software, optics, electronics, and other components. Furthermore, the research highlighted another challenge related to the engineers knowledge. Typically, automotive software engineers rarely possess a comprehensive understanding of the entire system, and face the challenge of dealing with numerous interconnected parts, diverse technologies, and the need to ensure compatibility with legacy software.

Düllmann et al. proposed the use of a model-driven domain-specific language (DSL) framework to facilitate the definition and analysis of CI/CD pipelines [22]. Their approach employs a model-based language, notably Business Process Model and Notation (BPMN), for creating visual representations of the pipeline outcomes. Nevertheless, the applicability of their framework is constrained to Jenkinsfile, the designated file format utilised for constructing and evaluating CI/CD pipelines within the Jenkins tool. Our experience has revealed a notable challenge when configuring Jenkins to operate seamlessly across diverse operating systems, including Windows.

Zampetti et al. conducted a comprehensive study on 8,000 non-forked projects sourced from GitHub, which employed at least one CI/CD pipeline [23]. They analysed the commits made to these projects, aiming to identify the restructuring or refactoring patterns performed on the pipelines. To achieve this, they employed a metric extractor capable of extracting 16 types of metrics, including total number of jobs, number of jobs permitted to fail, etc. The study by Zampetti et al. serves as a valuable complementary resource to our work. By leveraging their findings, we could further enhance automation and reduce manual development efforts by creating a pipeline that automatically adapts its structure based on those metrics.

Düllmann et al. presented a study that proposes the application of DevOps practices to the engineering of CD pipelines [11]. The authors emphasised that although CD

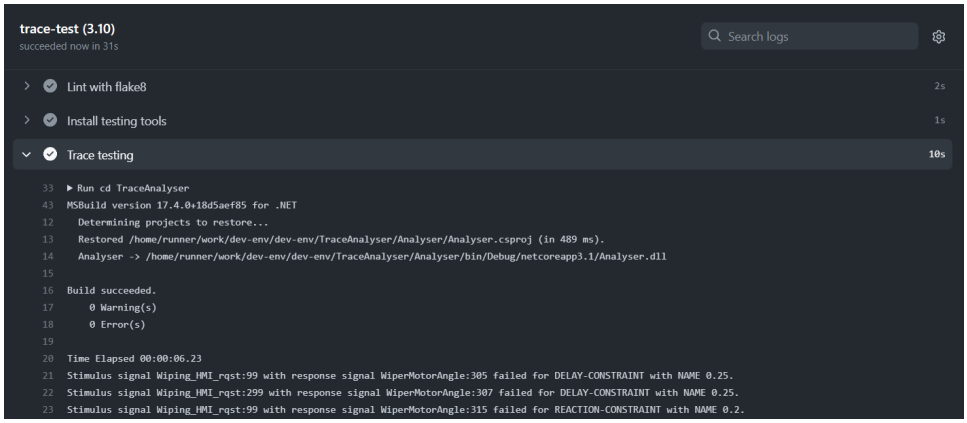


Fig. 6: Output from GitHub Actions pipeline.

pipelines are an integral part of DevOps practices, there is a lack of researches focusing on the engineering of the pipelines themselves. They argued that eliciting best practices could yield significant benefits for the CD pipelines, such as enhanced security and reliability.

VII. Conclusion and future work

We discussed the inclusion of a CI/CD pipeline to a model-based methodology that facilitates the development and architectural exploration of automotive system designs with temporal awareness. The proposed CI/CD pipeline included several improvements over traditional pipelines. These improvements include the use of parallel execution of jobs and a distributed build (where each job in the pipeline act as a unique node in the system) to satisfy performance and heterogeneous platforms requirements, respectively. We validated our work using two use cases namely break-by-wire and wiper motor automotive functionalities. Eventually, we discussed the challenges encountered during the design and development of the pipeline.

Future work may encompass several directions. One possible direction is to enhance the proposed pipeline using the metrics elicited by Zampetti et al. [23]. Another possible direction could focus on the quantitative validation of the CI/CD pipeline employing further use cases from the automotive domain. Eventually, future work may investigates how to containerise the different tools composing MoVES with, for instance, Docker technology.

Acknowledgements

The work in this paper has been supported by the Swedish Knowledge Foundation (KKS) through the Modev project.

References

[1] A. Bucaioni, S. Mubeen, F. Ciccozzi, A. Cicchetti, and M. Sjödin, "Modelling multi-criticality vehicular software systems: evolution of an industrial component model," *Software and Systems Modeling*, vol. 19, 2020.

[2] D. C. Schmidt *et al.*, "Model-driven engineering," *Computer-IEEE Computer Society-*, vol. 39, 2006.

[3] A. Bucaioni, A. Cicchetti, and M. Sjödin, "Towards a metamodel for the rubus component model." in *ModComp@ MoDELS*. Citeseer, 2014.

[4] A. Bucaioni, S. Mubeen, A. Cicchetti, and M. Sjödin, "Exploring timing model extractions at east-adl design-level using model transformations," in *2015 12th International Conference on Information Technology-New Generations*. IEEE, 2015.

[5] R. Eramo and A. Bucaioni, "Understanding bidirectional transformations with tggs and jtl," *Electronic Communications of the EASST*, vol. 57, 2013.

[6] A. Gamatié, S. Le Beux, É. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser, "A model-driven design framework for massively parallel embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, 2011.

[7] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, and T. Vardanega, "Chess: a model-driven engineering tool environment for aiding the development of complex industrial systems," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012.

[8] A. Bucaioni, S. Mubeen, J. Lundbäck, K.-L. Lundbäck, J. Mäki-Turja, and M. Sjödin, "From modeling to deployment of component-based vehicular distributed real-time systems," in *2014 11th International Conference on Information Technology: New Generations*. IEEE, 2014.

[9] R. Eramo, V. Muttillo, L. Berardinelli, H. Bruneliere, A. Gomez, A. Bagnato, A. Sadovykh, and A. Cicchetti, "Aidoart: Ai-augmented automation for devops, a model-based framework for continuous development in cyber-physical systems," in *2021 24th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2021.

[10] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *Ieee Software*, vol. 33, 2016.

[11] T. F. Düllmann, C. Paule, and A. van Hoorn, "Exploiting devops practices for dependable and secure continuous delivery pipelines," in *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*, 2018.

[12] A. Bucaioni, L. Addazi, A. Cicchetti, F. Ciccozzi, R. Eramo, S. Mubeen, and M. Sjödin, "Moves: A model-driven methodology for vehicular embedded systems," *IEEE Access*, vol. 6, 2018.

[13] V. R. Basili, "The experimental paradigm in software engineering," in *Experimental Software Engineering Issues: Critical Assessment and Future Directions: International Workshop Dagstuhl Castle*. Springer, 2005.

[14] A. Bucaioni, E. Ferko, and H. Lönn, "Trace-based timing analysis of automotive software systems: an experience report," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2021.

[15] EAST-ADL association, "East-adl v2.1.12," http://east-adl.info/Specification/V2.1.12/html/index.html, accessed: 2023-06-30.

[16] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study," *Computer Science and Information Systems*, vol. 10, 2013.

[17] L. Zhu, L. Bass, and G. Champlin-Scharff, "Devops and its practices," *IEEE software*, vol. 33, 2016.

[18] "Competitve advantage through devops - improving speed, quality, and efficiency in the digital world," https://inthecloud.withgoogle.com/hbr-report-19/hbr-research-report-google-cloud.pdf, accessed: 2023-06-30.

[19] "About github-hosted runners," https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners, accessed: 2023-06-30.

[20] A. W. Services, "Instance types," https://aws.amazon.com/ec2/instance-types/, accessed: 2023-06-30.

[21] L. E. Lwakatare, T. Karvonen, T. Sauvola, P. Kuvaja, H. H. Olsson, J. Bosch, and M. Oivo, "Towards devops in the embedded systems domain: Why is it so hard?" in *2016 49th hawaii international conference on system sciences (hicss)*. IEEE, 2016.

[22] T. F. Düllmann, O. Kabierschke, and A. Van Hoorn, "Stalkcd: A model-driven framework for interoperability and analysis of ci/cd pipelines," in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2021.

[23] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, "Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021.