Note on the Synchronization of Configurations between Digital Twin and Operational Industrial Environments

Petar Rajković[1,*,†], Dejan Aleksić[2,†] and Dragan Janković[1,†]

[1] *University of Niš, Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18104 Niš, Serbia*
[2] *University of Nis, Faculty of Sciences and Mathematics, Department of Physics, Višegradska 33, 18106 Niš, Serbia*

Abstract
Digital Twin technology has revolutionized the monitoring and management of operational industrial environments. It offers the possibility for a dynamic and virtual representation of physical systems, regardless of the number of affected layers. To make the digital twin fully accurate, it is crucial to have synchronized data, configurations, and software versions with real systems. This paper examines the critical considerations necessary for the effective synchronization of configurations between digital twins and their corresponding operational environments forcing the accuracy in mirroring physical attributes. It highlights conflicts when the configuration change comes simultaneously from different operational environments and gives guidelines on resolving them. The research describes the usage of various synchronization strategies, including event-driven updates and periodic polling, to ensure that the Digital Twin accurately reflects the current state of the main production environment. Furthermore, the paper addresses the implications of synchronization on system performance and maintenance scheduling. In conclusion, this paper presents a set of rules for synchronizing configurations discussing the process adjustments needed for a more efficient environment. It proposes best practices for maintaining congruence by ensuring system reliability and aims to guide practitioners to enhance operational efficiency.

Keywords
Digital twin, Industry 4.0, Inter-system synchronization

1. Introduction

The advent of Digital Twin (DT) technology has revolutionized the monitoring and management of operational environments (OE), offering a dynamic and virtual representation of physical systems [1]. This concept has been proven an exceptional asset when developing software elements for complex industrial systems [2]. Since they span across several ISA95 levels [3], the DT serves as the near-ideal development environment, where the effect of changes in one level could be easily simulated and checked in the rest of the system [4]. This concept allows faster discovery of many potential issues in the initial stages of the update project. Any problem spotted in the development and test environment could be immediately fixed lowering the possibility to reach production undetected.

In the examined case, DT is represented as the set software instance in the cloud environment resembling the production. To make DTs worthy, one must ensure that they are in congruence with the OEs [5]. Depending on the position of the software level, different OEs could be deployed, ranging from development through testing and validation, to main or production environments. This organizational pattern indicates our research question [6] – how to keep all these environments synchronized and how to address conflicts when it comes to them.

SQAMIA 2024: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 9--11, 2024, Novi Sad, Serbia
[*] Corresponding author.
[†] These authors contributed equally.
petar.rajkovic@elfak.ni.ac.rs (P. Rajković); alexa@pmf.ni.ac.rs (D. Aleksić); dragan.jankovic@elfak.ni.ac.rs (D. Janković)
0000-0003-4998-2036 (P. Rajković); 0000-0002-5833-6194 (D. Aleksić); 0000-0003-1198-0174 (D. Janković)

This paper explores the critical considerations necessary for the effective synchronization of configurations in various environments. It highlights the importance of real-time configuration checks, accuracy in the mirroring process, and the challenges posed by system complexity. The research explores various synchronization strategies, including event-driven updates, and periodic polling, to ensure that the DT accurately reflects the current state of the main OE. It highlights conflicts when the configuration change comes simultaneously from different OEs. To manage the mentioned challenges, we took the approach based on GitFlow regularly used in the GitHub environment. The idea behind this is to treat each OE as one of the important long-lived branches. Changes in different OEs are reflected by the creation of fixing branches in the Git environment and then merged accordingly.

The choice of GitFlow branching strategies for this scenario was advocated by the fact that our research group consists of relatively small teams (less than ten people) working on medical information (MIS) and manufacturing execution systems (MES), where the projects are in the stage that new developments are mostly modular or plug-in type, so the level of structural work could be separated from the system core. In conclusion, this paper presents a set of rules for synchronizing configurations discussing the process adjustments needed for a more efficient environment. It proposes best practices for maintaining congruence by ensuring system reliability and aims to guide practitioners to enhance operational efficiency.

2. Related Work

The synchronization of configurations between DTs and OEs is a topic that is well-covered in the literature. Especially in the industrial system development community, this is a topic that bothers developers and deployment engineers, since the human factor is, in many cases, still unavoidable.

A study [7] demonstrates how the environment of a DT can be effectively modeled and kept in sync with the real one by adding digital proxies of the relevant elements of the environment to the models of the DT elements. This approach was applied to a case study of a smart room with sensors. This setup is like our Internet of Things (IoT) nodes [8], so the modeling of digital proxies helps establish full DT elements of the lowest ISA95 levels. Digital proxies are virtual representations of relevant environmental elements that are added to the models of the DT elements. The inclusion of digital proxies aims to enhance the fidelity of the DT model, making it a more accurate and responsive representation of the physical system. This approach allows for a more comprehensive and dynamic model that can adapt to changes in the physical environment, as elaborated in [9]. Since we have a setup with multiple environments, the paper [10] gave us an idea of how to design the simulations for specific parts of the industrial system.

Research [11] focuses on the area remarkably close to our research domain. It analyzes the usage of DTs for modeling and simulation in production systems. Besides the paper is mostly oriented to a digital model of a human-robot-operated manufacturing, it employs the concept of synchronization with production database to allow accurate representation of system dynamics. We applied this approach for the synchronization of configuration files, too.

The next important aspect of synchronization design is how frequent synchronizations should be and how it can increase cost and data traffic [12]. The study formulates and analyzes the problem using stochastic models. Three solution approaches are explored: State-independent policy, State-dependent policy, and Full-information solution. The authors discuss solving the synchronization problem through simulation, and an approximate periodic state-dependent policy is proposed, yielding near-optimal results. Having this analysis in mind, we aimed for state-dependent policy, since we focus on maintaining proper software configuration across different installations.

These examples illustrate various applications and considerations in synchronizing DTs with their physical counterparts, highlighting the challenges and solutions encountered in different scenarios. Furthermore, DTs are not limited only to daily operations in industrial systems, but also to prototyping [13]. Apart from Industry 4.0, DTs found their place in architectural engineering [14], healthcare [15], aerospace [16], and many others.

3. Digital Twin, Operational Environments, and GitHub - Correlation

The synchronization of configurations between DTs and OEs, in the industrial system development community, is a topic that bothers both developers and deployment engineers, since the human factor is, in many cases, still unavoidable [17] . In the cases when the software of higher ISA95 levels is being developed, the complexity of the process reaches the level of information systems (IS). This comes with all known methodical and organizational patterns for IS life cycle maintenance [18]. On one end, developers and researchers have their developing sandboxes, and in the customer site, complete testing, validation, and main (or production) environment will be in use. This setup makes an interesting "ecosystem" where changes and synchronization must be fully controlled. Also, it must be considered that within one environment, like a test, there are multiple layers with different software instances where changes in one level could affect another.

At one point, we realized that all these different environments could be treated like long-living flow branches in a code versioning system like GitHub [19], and we could try to apply some of the known strategies for branch management. A branch in GitHub is a parallel version of a repository [20], and it could be easily integrated into standard development environments[21]. It exists within the repository but does not impact the primary (or main) branch. Branches allow you to work freely without disrupting the "live" version. When you have made the desired changes, you can merge your branch back into the main branch to publish those changes. In summary, a branch provides an isolated area for developing features, fixing bugs, or experimenting with innovative ideas within a repository.

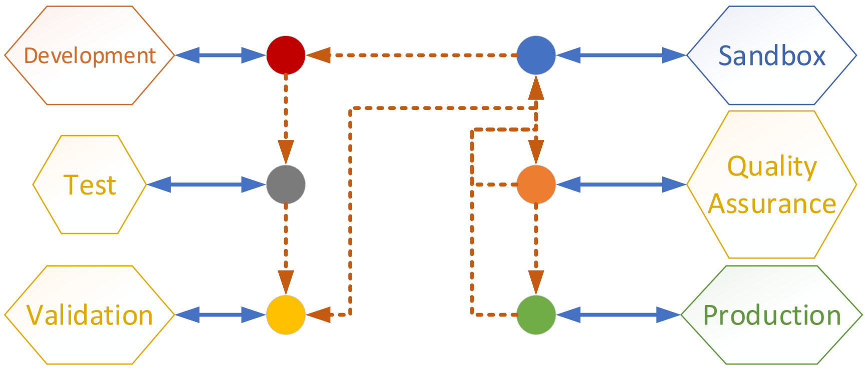

Figure 1: Standard interaction between different OEs (hexagons) and their DTs (circles)

GitHub is a developer platform that enables users to create, store, manage, and share their code. Git stands out from other version control systems (VCSs) since it allows users to easily modify and reorganize commit history. One can rebase or amend commits without affecting others' work. Next, creating branches is easy and fast. Git does not rely on sequential version numbers (like SVN), but, instead, it uses unique commit hashes, making it more flexible and adaptable. This led to our solution to maintain separate branches for every OE and to introduce a piece of software that will monitor changes in every environment and create necessary subbranches **Figure 1**. In the end, branches will be merged in Git, and then the updated version will be pushed back. This way ensures stability and the ability to alarm in case of changes. GitHub branching strategies are essential for managing codebases and ensuring efficient collaboration. More details on each of them can be found in [22]. Each of them has pros and cons and due to the size of our teams, the type of the projects, and the dynamics and type of the work. We choose to stick to GitFlow, as the basic branching strategy. The major challenges of GitFlow are the possibility of making the branch system too complex and thus slowing down continuous integration and delivery, while on the positive side a more efficient branching process, testing support, and handling several active versions of production versions [22].

One of the most important benefits of such an approach is the use of integrated code-merging mechanisms from the version control system and apply them to configurations in the repository. In that way, the use of the components of high fidelity is enabled, and the users can rely in the quality of the merge. The user will have to act only in cases when merge is not possible, which significantly reduces their engagement in the configuration maintenance.

4. GitFlow Scenario – Application, Adaptation, and Integration

As discussed before, in the examined setup – small development teams, several dozens of deployments, and an elevated level of involvement of the engineers on the customer side, GitFlow seems the most prominent, yet not ideal, choice. A small development team will result in the least number of development branches, which reduces the risk of a hard-to-manage and overcomplicated branching system. Testing within each branch could be easily done independently, and the integration testing could be easily pushed during the merge. Currently, most of our projects are in stable phases of the lifecycle. The development and update of new features and elements are strongly directed, so we can rely on every type of GitFlow branch for specific tasks.

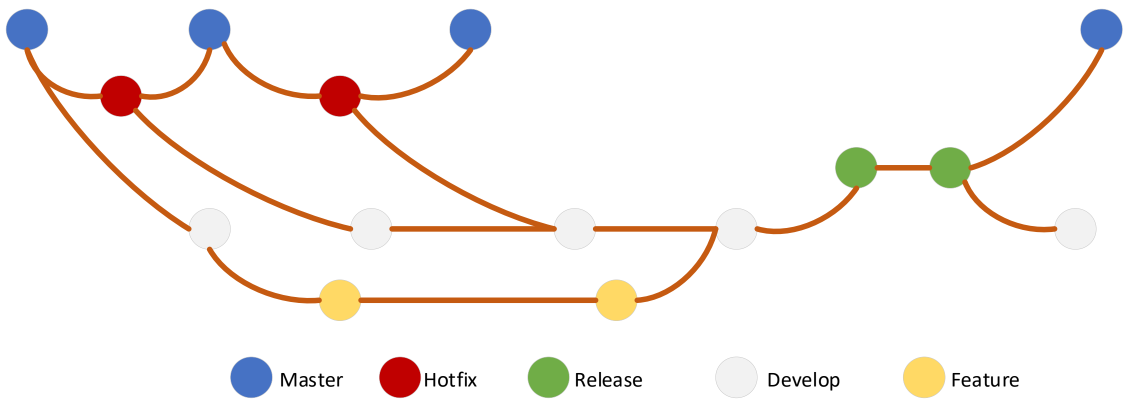

Figure 2: Standard interaction between different GitFlow branch types [23]

GitFlow [23], as one of the GitHub branching strategies is essential for managing codebases and ensuring efficient collaboration. The key point in this strategy is splitting branches into two groups – main branches and supporting branches. The main branches are those named Master and Develop. The Master branch is used to store the official release – in our case, it contains the configuration setup at the moment when the new software version is deployed. The develop branch is used as the integration branch when new features are developed. All developers' work will be merged into the Develop branch. Along with two main, there are also three supporting types of branching – feature, release, and hotfix. Feature branches are created from the develop branch and used when creating new features. Release branches function as gatekeepers to production, while the hotfix branches are used to address urgent issues. The usual workflow under GitFlow looks like the following [23] (**Figure 2**) - Developers work on feature branches, When stable, features merge into the Develop branch, Develop is periodically merged into Master for releases, and Tags mark specific release versions.

Since the main idea is to use GitFlow to maintain configurations we started with direct application of the strategy. Develop branch is set up to keep the configurations that are used in current development. They are up to some point different than the configurations of OEs since they need to be set up to support new features most efficiently. The developers were creating new feature branches, and as soon as they finished their work, they merged it back into the Develop branch. From this point of view, the sole development process looks easy to apply. Development teams, in our cases, both for MIS and MES are under ten people each, so the synchronization of parallel work is not an issue.

Supporting the configurations in the OE is a bit different story. If the work on development branches was clean and straight, for the OEs we cannot end up with a single Master branch that will support the production environment. In this case, we need to have separate Master branches for each OE. Since the idea was not only to support development but to actively maintain a DT, we need to store copies of the actual configurations on our side. Regarding OEs, we need to support at least two per deployment – test and production. Sometimes, there is also training and validation/staging, and sometimes even a sandbox for research and experimentation. To create a DT, we need to instantiate all services and clients for selected OE and distribute locally stored configurations. In that way, we ensure that the DT structure is maintained properly on our end.

As has been mentioned before, development is mostly a clear process. When the new feature has been developed, a release branch is formed and then the changes are merged into the test environment. From the test, they have been forwarded to other intermediate stages, and eventually to production. In case some bug has been discovered, the hotfix is applied to the Develop branch, and the entire process is repeated. In case some changes have been made in the production environment, it has been followed by a hotfix branch in the Master branch and then pushed back to Develop and then distributed to intermediates. For these two scenarios, the process could be mostly automated and there will be no need for some additional challenges and verifications. The cases when the configuration changes need to be validated are when the changes come from the intermediate OEs. To simplify the workflow, we organize the environment into the hierarchy, and always push the changes back to the Develop branch and then upstream to production if needed.

4.1. Adaptation – New Features

Direct use of GitFlow, in the case when multiple Master-type branches need to be maintained is not possible. For this reason, we needed to adapt the use-case scenarios. According to its name, the Develop branch should be used as the basis when new features are developed. This approach is regular for code files, but when it comes to configuration files, we realized that its role should be slightly different. In the scenario where the synchronization of configurations should be maintained, the Develop branch is used as the main backup source – the place from which "restore to factory settings" will take the data. To support the regular development process, at the start of the update project, we initially clone the Develop branch to a new one, called, by our convention SDC XYZ (where SDC stands for Service Delivery Contract, and XYZ is a contract number). Now, all feature branches will be created off the SDC branch and merged when development is complete. The SDC branch will be used then as the basis to push changes to all intermediate environments. When all testing and validation have been done, the SDC branch will be used to prepare Release branches and consecutively update the Master and the Develop branches.

4.2. Adaptation – Configuration Changes

During testing and validation, usually, many changes are about to happen. The users will experiment with some configuration setups until they define a satisfying environment. Here is important to notice that not all configuration changes are equally important and that not all branches that came out of these two processes will be promoted. In this stage, the changes are defined as feature branches, by default. They are kept as feature branches until the testing is running, and as soon as the change gets accepted, these branches are converted to Hotfix branches. Hotfix branches are then applied to adequate Develop branch and the change is pushed as usual. The remaining feature branches are kept as parallel branches as long as needed, and if not promoted to Hotfix branches will be discarded. The procedural change that we accepted here is that any change, regardless of its origin should be synchronized to the Develop branch and then pushed to other OEs **Figure 3**.

All production environment changes are always considered as Hotfix branches. Since they are mostly done by the engineers on the customer side, they are assumed as the changes of the highest priority and, after synchronization with the Master branch pushed to all other environments. This type of change is the most challenging to manage. Production environments are often under higher security regimes which means that access is not possible at any moment. Having this in mind it could happen that some changes become undetected from our end, thus never reaching the DT. The consequence is then seen in the next deployment where the change got overwritten with the latest version that comes from the release branch. To suppress such a problem, the complete system installation is enriched with additional file-watching and sniffer tools that send notifications back in the case of configuration change.

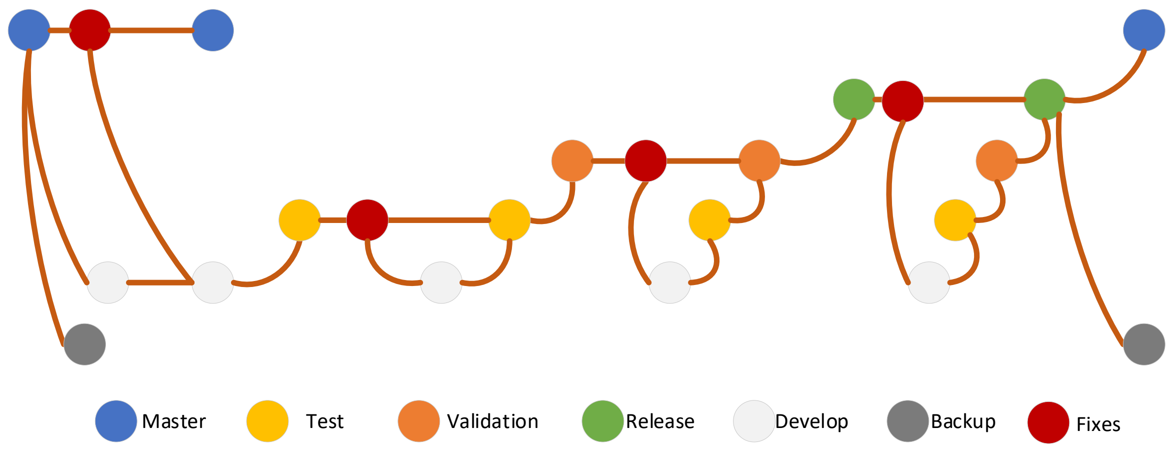

Figure 3: Fix propagation between different branches.

4.3. Integration

To make the complete process more effective, we had to develop several supporting tools that helped in maintaining DTs and ensuring consistency. This came as the requirement to monitor changes in production since it is not possible to fully rely on engineer's reports. Besides, they are inaccurate in a small percentage of cases, it is more probable that they came too late, or not be properly processed on our end.

The first supporting tool is a file system watcher-based service. It monitors the configuration folders and creates reports with any change that happens. Each of these reports is sent back to the DT, making this scenario the typical event-driven synchronization (**Figure 4**, left). The other tool, running in DT gateway, report analyzer, takes the received report and based on its origin and change type creates a new branch request. Branching requests could be created either as a hotfix or feature branch. These requests could be either kept in the report analyzer and waiting for the user for approval or immediately pushed to the GitHub and created branch.

Branch management is done in some of the commercially available branch management tools. With this approach, we ensure that already established merge routines are followed in the expected way and that the quality of merged code will be at the expected level. The last of the tools that we employ is configuration management which is used when new sets of configurations need to be packed and shipped to the end location. Since they are already used in the deployment process for industrial systems, in this case, they are just slightly modified to create software node images in a DT structure [24].

On the other hand, we added periodic polling wherever possible (**Figure 4**, right). In cases where some of the intermediate environments, or even our development servers, have access to the customer's production, we send enriched ping messages with the hash code generated by the configuration content. If the destination server answers with a different hash, this would trigger the alarm on our end and start the necessary configuration check. The polling mechanism is the standard part of any MIS, MES, or Enterprise Resource Planner (ERP) client implementation thus its enrichment is a logical enhancement task. When sent from the server, it is not every ping message intended to initiate a check on the client end. The period for enriched ping could be configured and usually, this would be daily. Changes in the client side will raise the flag to recalculate the hash and then, the next ping will serve with the change notification, which will lead to the change synchronization process.

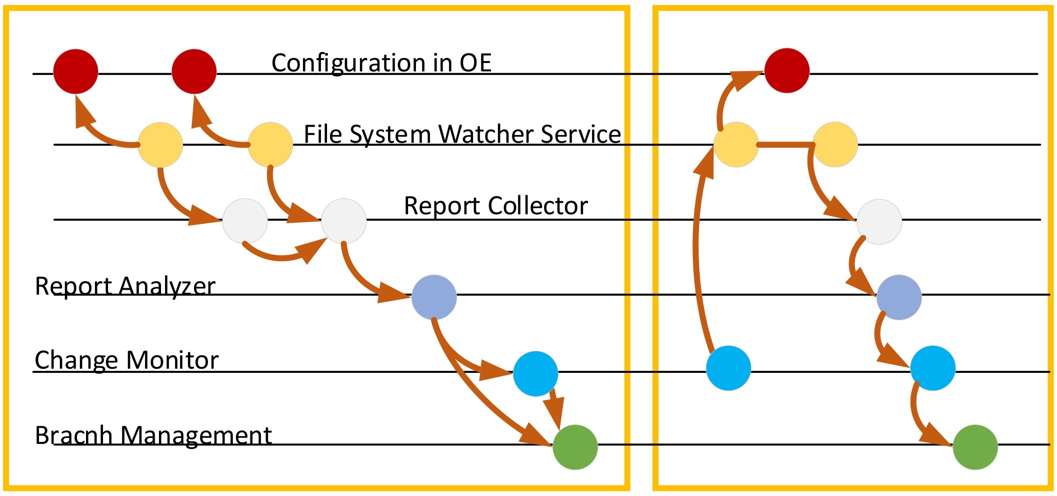

Figure 4: Data flow between configuration changes and supporting tools (left – event-driven change registration, right – change registered after polling).

5. Discussion

Keeping DTs in an accurate state is necessary when comes to complex system maintenance. In that sense, the three types of consistency that need to be ensured, in order of importance - codebase, configuration, and data consistency. Codebase consistency is the most important part since all functionalities depend on them. In the second place is configuration consistency, because the system behavior should be fully replicable in any domain. For some systems, data consistency is important since they need to keep all environments equal. From the technical point of view, ensuring codebase and data consistency is a bit easier case since the changes are generated in one place and then distributed across the environment. For codebase this is the distribution from the development environment through tests to production, and for data consistency is vice versa.

The most challenging part, and the focus of our work, is the configuration consistency since they could be changed in any segment of the complex environment, and then need to be distributed correctly to other environments. To ensure configuration consistency, we decide to use GitFlow branching strategies and to treat each configuration change as the potential branch. It is important to state that GitFlow is not considered "the best" strategy with code developers. Nowadays it is mostly replaced with Trunk-Based development, but due to certain advantages, it has been chosen to support the maintenance of system configurations. The biggest pros are strict code quality control and isolated feature development which are the most important in the examined case. Longer-lived branches, observed as the "con" generally are very useful here, since they represent non-productive environments.

Since our projects have been developing for many years, we have a lot of different change requests and SDCs that vary in scope and size significantly. Looking at deployment history we could identify 23 SDCs and 56 change requests which has been done in MIS since 2008. All of them are done without integration with Git. The concept of using Git is new for us, and we currently have only one SDC completed with the support of Git, and one in progress. For this reason, we cannot define more comparison scenarios, than the ones whose results are shown in Table 1. Thus the word "Note" in the title of the paper. To make comparable SDCs we choose two that are contextually connected. The older one, named 2023Q3 had an integration with the external systems in its focus, and the newer one supports fine-tuned synchronization processes that aim to fully control the integration. In total, both targeted the same set of data exchange processes and the sets of configurable parameters are close by its size. With the help of additional tools, we managed to reduce the number of configuration problems. Table 1 shows the results for two SDCs of comparable size – named 2023Q3 and 2024Q1. Both have around 120 configuration changes that were created in the test environment, around 40 in test, and around 10 in production. The difference is that 2023Q3 was developed without the support of the suggested branching and notification system.

Table 1
Problems with configuration synchronization in two similar SDCs

Project	Environment	Total changes	Undetected	Overwritten	Wrongly Merged	Failure rate
2023Q3	Test	129	11	14	12	20.93%
2023Q3	Validation	41	7	3	4	34.14%
2023Q3	Production	11	4	1	0	45.45%
2024Q1	Test	117	5	6	1	10.25%
2024Q1	Validation	46	2	3	1	13.04%
2024Q1	Production	8	0	1	0	12.5%

Without a dedicated system to detect changes, we used to spend a lot of time comparing sets of files, with an exceedingly high probability of missing the change, overwriting it, or being wrongly merged. These problems are especially problematic in production leading to unnecessary re-deployments. With the introduction of branching strategies and a notification system, we managed to fully avoid the number of undetected changes in the production. Since each change is detected, they tend to be ridiculously small, thus the probability that they could be wrongly merged is insignificant.

When talking about undetected changes, in 2024Q1 they come as the consequence of the bugs discovered in notification tools, while the problem with overwritten changes appeared in cases when multiple configuration files were copied simultaneously from various locations. The additional benefit, that we were not able to measure, is the time spent on comparison between files in various locations. During the 2024Q1 project, we only need to make such a change when deploying to production, since this is the part of business process. Apart from this, we make such checks only a few times in test and validation when the problem of undetected change appears.

6. Conclusion

Developing MIS, MES, and ERP client systems members of our research group found themselves in situations where changes have been created in some of the environments, being undetected and then not distributed or overwritten. To overcome such a problem, we decided to use the features of existing GitHub systems to maintain different versions of configurations which are then used in DTs. Since multiple environments could generate changes, an appropriate branching strategy had to be used. Evaluating the standard flows, we came across GitFlow which seemed to be adequate for our setup, taking into account the number of programmers, the number of deployments, and several nodes in different DTs. This approach is then adapted on our end and enriched with additional support tools.

On the other side, we must state that this paper shows only the initial results of our approach. We applied our update to a limited set of data, thus could not be considered to be fully accurate, but promising. Having in mind the previous experience and evaluated cases, we can define the following set of guidelines to improve configuration maintenance:

- Use code version system (CVS) to maintain configurations across different environments. DTs in separate environments could be used for testing, but CVS will rigidly indicate any change.

- Use separate, long-living branches for each OE. Any change in production is to be treated as an immediate hotfix and then pushed to the code repository. Any change in other OEs to be treated as feature branches and validated as any regular change from development.
- Any change, regardless of the point of origin should be pushed back to test and develop and then gradually merged back to production.
- Any bug, issue, or problem needs to be validated in a different environment and then scheduled for an update according to its priority.
- Include supporting tools to detect changes in various environments and periodically compare them with the configurations in the repository.
- Configurations are not the code, per se, but they have a huge influence on system work. Thus, support new configurations with appropriate test methods whenever possible.

Eventually, we managed to reduce the number of undetected and erroneous configurations in every environment leading to higher customer satisfaction and a more efficient software lifecycle.

Acknowledgments

The Ministry of Science, Technological Development and Innovation of the Republic of Serbia supported this work [grant number 451-03-65/2024-03/200102].

This work is partially supported by the cost action CA 19135 CERCIRAS (Connecting Education and Research Communities for an Innovative Resource Aware Society).

References

[1] D. Jones, C. Snider, A. Nassehi, J. Yon, B. Hicks, Characterising the Digital Twin: A systematic literature review, CIRP J. Manuf. Sci. Technol. 29 (2020) 36–52. doi:10.1016/j.cirpj.2020.02.002.

[2] F. Tao, H. Zhang, A. Liu, A. Y. C. Nee, Digital twin in industry: state-of-the-art, IEEE Trans. Ind. Inform. 15.4 (2019) 2405–2415. doi:10.1109/tii.2018.2873186.

[3] ISA95, enterprise-control system integration URL: https://www.isa.org/standards-and-publications/isa-standards/isa-standards-committees/isa95.

[4] Y. Qamsane, C.-Y. Chen, E. C. Balta, B.-C. Kao, S. Mohan, J. Moyne, D. Tilbury, K. Barton, A unified digital twin framework for real-time monitoring and evaluation of smart manufacturing systems, in: 2019 IEEE 15th international conference on automation science and engineering (CASE), IEEE, 2019. doi:10.1109/coase.2019.8843269.

[5] K. T. Park, S. H. Lee, S. D. Noh, Information fusion and systematic logic library-generation methods for self-configuration of autonomous digital twin, J. Intell. Manuf. (2021). doi:10.1007/s10845-021-01795-y.

[6] Y. Han, D. Niyato, C. Leung, D. I. Kim, K. Zhu, S. Feng, S. X. Shen, C. Miao, A dynamic hierarchical framework for iot-assisted digital twin synchronization in the metaverse, IEEE Internet Things J. (2022) 1. doi:10.1109/jiot.2022.3201082.

[7] J. A. Llopis, J. Criado, L. Iribarne, P. Muñoz, J. Troya, and A. Vallecillo, "Modeling and Synchronizing Digital Twin Environments," 2023 Annual Modeling and Simulation Conference (ANNSIM), Hamilton, ON, Canada, 2023, pp. 245-257.

[8] P. Rajković, D. Aleksić, D. Janković, The Implementation of Battery Charging Strategy for IoT Nodes, in: Lecture Notes in Computer Science, Springer Nature Switzerland, Cham, 2024, pp. 40–51. doi:10.1007/978-3-031-48803-0_4

[9] B. Tekinerdogan, On the Notion of Digital Twins: A Modeling Perspective, Systems 11.1 (2022) 15. doi:10.3390/systems11010015.

[10] M. Bonney, P. Gardner, D. Wagg, R. Mills, Case Study Of Connectivity of Digital Twins and Experimental Results, in: 8th International Conference on Computational Methods in Structural Dynamics and Earthquake Engineering Methods, Athens, 2021. doi:10.7712/120121.8569.19285.

[11] A. Kampa, Modeling and Simulation of a Digital Twin of a Production System for Industry 4.0 with Work-in-Process Synchronization, Appl. Sci. 13.22 (2023) 12261. doi:10.3390/app132212261.
[12] B. Tan, A. Matta, The digital twin synchronization problem: framework, formulations, and analysis, IISE Trans. (2023) 1–25. doi:10.1080/24725854.2023.2253869.
[13] M. Yazdi, Digital twins, and virtual prototyping for industrial systems, in: Springer series in reliability engineering, Springer Nature Switzerland, Cham, 2024, pp. 155–168. doi:10.1007/978-3-031-53514-7_9.
[14] Z. Ghorbani, M. Cramer, J. Messner, Operational digital twins: definition and common use cases, in: ASCE international conference on computing in civil engineering 2023, American Society of Civil Engineers, Reston, VA, 2024. doi:10.1061/9780784485231.015.
[15] E. Katsoulakis, Q. Wang, H. Wu, L. Shahriyari, R. Fletcher, J. Liu, L. Achenie, H. Liu, P. Jackson, Y. Xiao, et al., Digital twins for health: a scoping review, NPJ Digit. Med. 7.1 (2024). doi:10.1038/s41746-024-01073-0.
[16] L. Li, S. Aslam, A. Wileman and S. Perinpanayagam, "Digital Twin in Aerospace Industry: A Gentle Introduction," in IEEE Access, vol. 10, pp. 9543-9562, 2022, doi: 10.1109/ACCESS.2021.3136458
[17] J. Vatn, Industry 4.0 and real-time synchronization of operation and maintenance, in: Safety and reliability – safe societies in a changing world, CRC Press, 2018, pp. 681–686. doi:10.1201/9781351174664-84.
[18] A. Lanz, B. Weber, M. Reichert, Time patterns for process-aware information systems, Requir. Eng. 19.2 (2012) 113–141. doi:10.1007/s00766-012-0162-3.
[19] V. Cosentino, J. L. Canovas Izquierdo, J. Cabot, A systematic mapping study of software development with github, IEEE Access 5 (2017) 7173–7192. doi:10.1109/access.2017.2682323.
[20] Version control basics | dbt Developer Hub. URL: https://docs.getdbt.com/docs/collaborate/git/version-control-basics.
[21] GitHub extension in VS 2017. URL: https://www.c-sharpcorner.com/article/install-and-use-github-extension/.
[22] What is the best Git branch strategy? | Git Best Practices. URL: https://www.gitkraken.com/learn/git/best-practices/git-branch-strategy.
[23] A successful Git branching model. URL: https://nvie.com/posts/a-successful-git-branching-model/.
[24] P. Rajković, D. Aleksić, A. Djordjević, D. Janković, Hybrid Software Deployment Strategy for Complex Industrial Systems, Electronics 11.14 (2022) 2186. doi:10.3390/electronics11142186.