Da Silva, J., Illarramendi, M., Iriarte, A. (2023). Using Runtime Information of Controllers for Safe Adaptation at Runtime: A Process Mining Approach. In: Guiochet, J., Tonetta, S., Schoitsch, E., Roy, M., Bitsch, F. (eds) Computer Safety, Reliability, and Security. SAFECOMP 2023 Workshops. SAFECOMP 2023. Lecture Notes in Computer Science, vol 14182. Springer, Cham. https://doi.org/10.1007/978-3-031-40953-0_8

Using Runtime information of controllers for safe adaptation at runtime: a Process Mining approach

Jorge Da Silva[1], Miren Illarramendi[1], and Asier Iriarte[1]

MGEP Mondragon Goi Eskola Politeknikoa S. Coop
{jorge.dasilva,asier.iriarte}@alumni.mondragon.edu and
millarramendi@mondragon.edu

Abstract. The increasing complexity of current Software Systems is generating the urge to find new ways to check the correct functioning of models during runtime. Runtime verification helps ensure that a system is working as expected even after being deployed, essential when dealing with systems working in critical or autonomous scenarios. This paper presents an improvement to an existing tool, named CRESCO, linking it with another tool to enable performing periodical verification based on event logs. These logs help determine whether the functioning of the system is inadequate or not after the last periodic check. If the system is determined to be working incorrectly, new code files are automatically generated from the traces of the log file, so they can be replaced when a faulty scenario is to occur. Thanks to this improvement, the CRESCO components are able to evaluate their correctness and adapt themselves at runtime, making the system more robust against unforeseen faulty scenarios.

Keywords: Runtime adaptation · Process Mining · Runtime fault detection.

1 Introduction

Testing, validation and adaptation of deployed software is an important necessity, due to the increasing complexity and variety of current Software Systems (SS). Given that the consequences of failure of these systems can turn into a catastrophe, ensuring that SS operation is always as expected is essential, especially for systems located in critical scenarios. If the functioning is not the expected one, it is also important to provide a method which helps the system return to its expected working flow.

One of them is verification, which is a dynamic analysis technique used to determine whether the running behaviour of a system satisfies a previously defined correctness property, as stated in [13]. It can be obtained by following a Model Driven Engineering (MDE) approach. The application of this technique is beneficial because it reduces the learning curve by designing systems using graphical elements that directly link to a familiar domain. It also lets a broader range of

subject experts to understand the system and ensure that it meets the needs of the user [30]. Different techniques for designing MDEs exist, among which Unified Modelling Language State-Machines (UML-SM) constitute a widely spread formalism.

However, MDE techniques need to be integrated into a runtime perspective. To achieve that, one of the possibilities consists on following runtime verification and adaptation according to the models@runtime approach. As long-running SSs used in critical scenarios need to safely adapt to the changes of their execution environment, it is usually inconvenient to take them offline to apply such changes, so these actions need to be done at runtime with the least human intervention possible [7, 9].

In this paper, we present an approach to make a SS adapt automatically when failures are detected, avoiding malfunctioning and faulty scenarios. The main contribution provides the following benefits:

1. Automatic runtime adaptation is performed, thus when a failure is detected, there is no necessity for a developer to get involved to try solve the detected point of failure, providing the SS the capacity to run without supervision.
2. Adaptation is based on the previous functioning of the system: it will adapt to correct malfunctioning taking into account the previous valid executions.

The paper is organized as follows. Section 2 details the work's background. Section 3 details the work done during the study. In section 4 the related work is presented. Finally, conclusions and future work are provided in section 5.

2 Background

In this section, we will define some concepts that will help understand the work presented in this document.

One way to perform runtime verification is to observe the runtime information (traces) sent by the software controller to the externalized runtime checker/ adapter. Since correct traces will be finite and predefined in the checker/adapter, when the received trace is not defined as a correct one, the checker/adapter comes to a state that a Trace-Violation has been detected.

Runtime Adaptation is a technique prevalent to long-running, highly available software systems, whereby system characteristics (e.g., its structure, locality ...) are altered dynamically in response to runtime events (e.g., detected hardware faults or software bugs, changes in system loads), while causing limited disruption to the execution of the system [9].

In [15] is proposed an "externalized" runtime adaptation system that is composed of external components that monitors the behaviour of the software component of the running system. These external components are responsible for determining when a software component's behaviour is within the envelope of acceptable system parameters. When the software component's behaviour fall outside of the expected limits, the external components start the adaptation process. To accomplish these tasks, the externalized mechanisms maintain one

or more system models, which provide an abstract, global view of the running system, and support reasoning about system problems and repairs. DevOps is a development methodology aimed at bridging the gap between Development and Operations. To do so, emphasis is established on different aspects, such as communication and collaboration, quality assurance, continuous integration and automatically deployed delivery [22]. As stated in [6], "DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality". The DevOps methodology was created as a counterpart to the V-Model, where the process is linear and, once finished, does not feedback itself. In DevOps, instead, the process is continuous, as it can be observed in Fig. 1.

Tests done during the development of SW are tasks of high importance. In DevOps, however, the tests are not only done in the development phase, but also in the operation phase. Monitoring is performed as well during this phase, verifying the correct functioning of the SW, i.e., runtime verification is used to ensure the correctness of the system. This last process makes it possible for a safe adaptation to be performed whenever an error is detected, i.e., a runtime adaptation is carried out to correct the error.

One way to perform runtime verification in DevOps is by observing the runtime information – also known as traces – sent by a software controller to an externalized runtime checker, obtaining two different SSs. Correct traces will be finite and predefined in the checker. Therefore, when the received trace is not correct, the checker will notify that a Trace-Violation has been detected.

Process mining is a technique oriented towards discovering, monitoring and improving real processes using the event logs recorded by information systems [33]. One of the many tools that allows to perform process mining is ProM, which is developed by the Eindhoven University of Technology [2]. The logs employed act as the foundation for finding valuable patterns using different methods, one of which is the Directly Follows Graph (DFG). As explained in [25], this type of graph can be derived from the log and represents which activity follows another one directly.

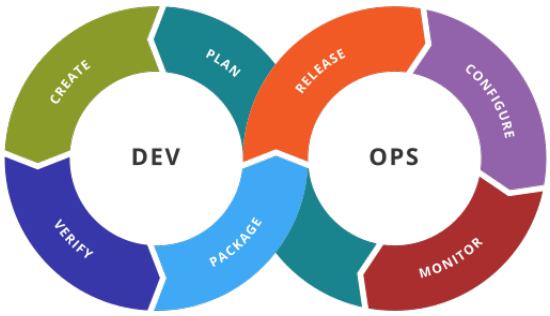

Fig. 1. Entire process of the DevOps methodology.[21]

As mentioned earlier, adapting at runtime is an important part of the DevOps methodology. To provide a high-quality runtime adaptation, one of the possibilities is to use automatic code generation, which is a method centered around creating executable code by an executing process. This technique permits programs to create specialized instruction sequences based on the information obtained at runtime [12]. Embracing the usage of automatic code generation is significantly important, as it enables the code to optimize itself, obtaining better results than with only once compiled programs.

This work is based on [20]. In this work, a framework named CRESCO is used for the generation of reflective software components, which provide information regarding the internal status of UML-SM element-based components. These components communicate with a Runtime Safety Properties Checker (RSPC), which ensures that the observable actions to be taken by the CRESCOs are valid. If they are not, the CRESCO machines are warned.

In the implementation of this work, we have dockerized the framework to easily use it in different use cases (both CRESCO and the RSPC).

2.1 The CRESCO framework

The CRESCO framework is the C++ implementation of the RESCO metamodel [19]. RESCO is a metamodel composed of two packages: (1) a design package oriented towards modeling the application-specific part and (2) a runtime package that enables a UML-SM based software component to reflect the model it comes from.

Any specific application is modeled using the design package and its main goal is to describe a State Machine (SM). A component named the Executor contains the implementation of the reactions that need to be triggered whenever an action occurs in the SM. Both the SM and the Executor components are generated automatically from the UML-SM model defined by the designer (i.e. by the Acceleo tool [1]).

The runtime part, which is generic for all components and applications created with RESCO, is modelled using the runtime package. It includes generic elements oriented towards providing runtime model observation capabilities. The runtime elements are used for implementing the execution semantics of state machines.

3 Proposed approach

The aim of the following subsections is presenting the details of the process of periodically checking the correctness of the UML-SM components, as well as the steps to adapt their behaviour dynamically whenever an unexpected situation is addressed.

3.1 Periodical verification of the CRESCO components

Fig. 2 presents a graphical representation of the process and the tools involved on it.

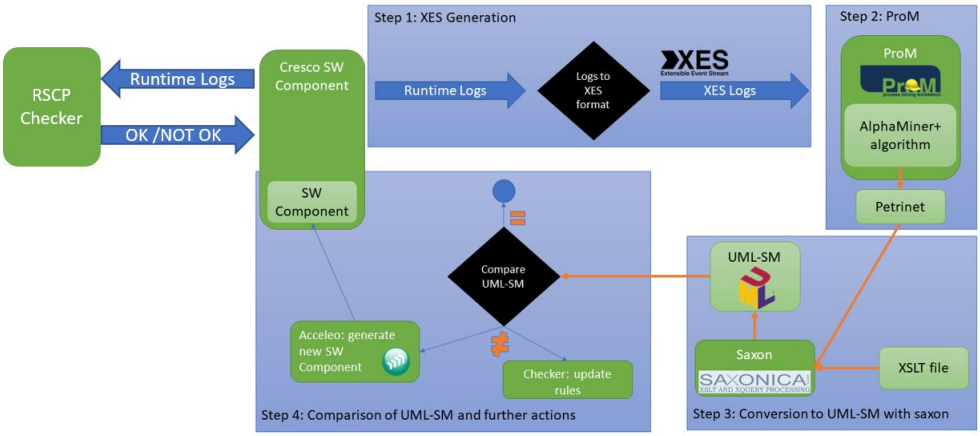

Fig. 2. Process for runtime adaptation of the CRESCO SS

First, each CRESCO component periodically sends the log where the transitions performed by the component are stored to an external program. Then, the log format is adapted to the one needed to import the information to ProM (Extensible Event Stream (XES)), which is a standard defined for specifying, as stated in [16].

The second step is performed using the ProM tool. The previously adapted log is imported to this tool, and it transforms the logs to a PetriNet. Even if ProM can generate more types of diagrams, some of those cannot be exported, or the data displayed is not adequate to be used later. However, using the AlphaMiner+ algorithm to generate the PetriNet, it is possible to attain something similar to a UML-SM with what the log represents. This is because the AlphaMiner+ algorithm is based on the directly follows graph (DFG). A DFG is a graph where activities (nodes) and its directly-follows relationships (directed edges) are represented.

This can be translated to UML-SMs, as the nodes and directly-follows relationships of the DFG will resemble the states and transitions of the UML-SMs, respectively.

After the PetriNet has been generated, it is necessary to conduct a transformation, so a properly generated UML-SM is obtained. For that, a Extensible Stylesheet Language Transformation (XSLT) file is created. The purpose of this type of files is to enable the transformation of XML files into other XML files. This was achieved with saxon [29].

The obtained UML-SM is then compared to the one currently running, to ensure that the transitions between states remain the same. If a change in the relations of the states of the UML-SM is detected, the Acceleo [1] tool is used to generate the new files of code that will rule the CRESCO machine. The Acceleo tool takes as input the new State Machine diagrame and the CRESCO framework, automatically generates the aforementioned files of code. This process is described in [19].

All the code files generated are stored to be used when an unexpected or erroneous situation is detected and is necessary to correct the functioning of the CRESCO machine.

3.2 Runtime adaptation upon detection of unexpected situations

Runtime adaptation constitutes a possible method to be used when ensuring the maintenance of a SS. Ensuring that the SS retain their continuous execution is essential.

In the current solution, the approach followed consists on rebuilding the CRESCO machines and the checker whenever an unexpected erroneous situation is detected. To do that, both the CRESCOs and the checker are replaced by the newly generated code files, which are stored in a folder. Afterward, the new deployment of the systems is activated and initialized from the point where the error was detected. This is to ensure that the flow of the CRESCO systems is continued through different executions.

3.3 Result

A video with an example has been prepared to show the process. The example is based on a Train Traction and Door controller (Fig.3) and it can be found on https://github.com/millarramendi/Safecomp.git

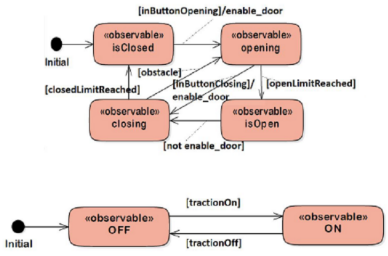

Fig. 3. Door and Traction UML-SM controllers

4 Related work

Several other successful works exist in the frame of runtime adaptation. Thus, it is the goal of this section to briefly highlight the features of those works and their differences with the work presented in this paper.

Creating a runtime checker for checking system level safety properties is an important research problem. Existing approaches to runtime verification require specifications that not only define the property to monitor, but also contain details of the implementation, sometimes even requiring the implementation to add special variables or methods for monitoring [28], that is code instrumentation.

In [18], they defined a generic SW monitoring model and analyzed different existing monitors. Depending on the programming language used and the formalism used to express the properties, different implementations of monitors have been proposed [24], among others CoMA [5], RV-MONITOR [10] and AspectC++ [32], BEE++ [8], DN-Rover [11], HiFi [4] etc. The abstraction level of specification and monitoring of the above solutions is the same: specifications and the monitor's checking properties are at the component or class level.

One of the first works exploring the idea of generating monitors from a specification and developing a tool chain for it was the MaC (Monitoring and Checking) framework [23]. They presented a tool that uses the specification information to generate the checker. It is a debugging tool, designed to supplement testing. While the MaC framework showed the effectiveness of monitoring systems for compliance with specification, the framework's target was primarily for software systems with instrumented code, requiring modifications to the source code and making it unsuitable for usage in safety-critical embedded applications. In contrast to our approach, this framework is not isolated from the target system.

In the same direction, ConSerts [31] framework monitors the system's conformance with its certificates during execution. The solution is focused on the Safety Integrity Level (SIL) of the different components that are part of the system. This approach works at a system requirement level and not in detecting errors. In this regard, the approaches might be complementary to the present work.

ModelPlex [27], checks cyber-physical systems (CPSs) in model terms. It uses the KeYmaera proof engine to verify the safety of a CPS model, and generates monitors to check system traces for compliance with the model. It requires a high degree of user expertise both to write the original model in Dynamic Logic (the KeYmaera mathematical language) and to guide the proof engine towards a formal proof of the property. Our approach is a solution designed to check system level safety properties in model terms but without required expertise in formal methods.

LuMiNous [28] framework's target is the same area as the checker presented in this paper: system level specifications are checked by components' level information. The contribution of this framework relies on the translation of high-level specifications into runtime monitors but, in this case, their solution is for Java (AspectJ based solution), which is not suitable for embedded systems.

In [14], an idea is proposed to achieve fault tolerance in critical features of software functionalities of modern cars, using the Plug&Play technique to integrate new components in a Runtime Environment (RTE), which contains safety-related mechanisms like health monitoring.

The Plug&Play methodology explained in [14] is specifically designed to achieve correctness in critical systems thanks to its Electrics and Electronics (E/E) architecture, which provides built-in mechanisms to achieve fault tolerance. However, the main intention of the technology is to have the possibility to add new functionalities to a complex system (in this case a car).

In [26], a solution is proposed to tackle divide-by-zero and null-deference errors, where a component for *recovery shepherding* detects these errors and its further occurrences during the execution of a program and avoids them to ensure the correct functioning of the application even after the failure happened.

The proposed solution, however, only takes into account two errors, as the subsequent errors that are corrected are also of one of those types. Applying this methodology in a critical system could be dangerous, as the faulty scenario needs to be correctly analyzed and treated, and not only dodged, because otherwise the system could suffer unaffordable losses.

In [17], a lightweight solution to runtime error recovery is proposed, where the errors of interest are detected by the system, and then a copy of the state of the program before the buggy state is done, in order to check in a *sandbox-like* situation different solutions, picking the best of them to overcome the error.

Even though this solution provides an appropriate way to handle the errors generated during runtime, as the solution is only applied once (as this solution considers the context of the execution), this could lead to serious problems in critical scenarios, where faulty configurations need to be avoided the next time they happen without the need to redo the process again.

5 Conclusions and future work

The paper presents a periodical check process to verify the correctness of the UML-SM of a SS. In the case of detecting discrepancies between both UML-SMs, corrections of the currently running UML-SM are generated based on event logs, which are stored until a unexpected scenario happens. This adaptation is done on top of the CRESCO framework, which is designed to provide runtime observable components.

The main conclusion is that the defined methodology to adapt when a faulty scenario is detected is capable of maintaining the CRESCO machines working, retaining the expected functionality.

Another conclusion is that the process of periodically checking if the functioning of the CRESCO machine is correct based on the log of the UML-SM provides little overhead to the SS. This enables to effectively maintain a good response time, demonstrating that the method used is adequate for working in critical scenarios.

Some limitations in the process exist too. For example, as explained in [3], the naive use of DFGs may cause several problems:

- Possibility to lead to Spaghetti-like DFGs, generating loops even when activities are executed at most once.
- Information mapped onto DFGs can be misleading the average time reported between two activities as it needs to be taken into account that only the situations where they directly follow each other are considered.

Nevertheless, this limitation would be solved by selecting another algorithm that addresses the problems presented by DFGs, being adequate to be integrated into the process proposed.

In the future, we would like to evolve the method followed to achieve run-time adaptation, so that CRESCO software systems do not lose communication with their control system during run-time adaptation. This would make the solution much more realistic and suitable for efficient integration into CPS.

References

1. Acceleo. Tech. rep., https://www.eclipse.org/acceleo/ (2016)
2. Prom tools. Tech. rep., https://www.promtools.org/doku.php (2020)
3. van der Aalst, W.M.: A practitioner's guide to process mining: limitations of the directly-follows graph (2019)
4. Al-Shaer, E.S.: A hierarchical filtering-based monitoring architecture for large-scale distributed systems. Old Dominion University (1998)
5. Arcaini, P., Gargantini, A., Riccobene, E.: Coma: Conformance monitoring of java programs by abstract state machines. In: International Conference on Runtime Verification. pp. 223–238. Springer (2011)
6. Bass, L., Weber, I., Zhu, L.: DevOps: A software architect's perspective. Addison-Wesley Professional (2015)
7. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer **42**(10), 22–27 (2009). https://doi.org/10.1109/MC.2009.326
8. Bruegge, B., Gottschalk, T., Luo, B.: A framework for dynamic program analyzers. In: ACM SIGplan Notices. vol. 28, pp. 65–82. ACM (1993)
9. Cassar, I., Francalanza, A.: Runtime adaptation for actor systems. In: Runtime Verification. pp. 38–54. Springer (2015)
10. Daian, P., Guth, D., Hathhorn, C., Li, Y., Pek, E., Saxena, M., Şerbănuţă, T.F., Roşu, G.: Runtime verification at work: A tutorial. In: International Conference on Runtime Verification. pp. 46–67. Springer (2016)
11. Diaz, M., Juanole, G., Courtiat, J.P.: Observer-a concept for formal on-line validation of distributed systems. IEEE Transactions on Software Engineering **20**(12), 900–913 (1994)
12. Engler, D.R., Proebsting, T.A.: Dcg: An efficient, retargetable dynamic code generation system. ACM SIGPLAN Notices **29**(11), 263–272 (1994)
13. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. Engineering dependable software systems pp. 141–175 (2013)
14. Frtunikj, J., Rupanov, V., Camek, A., Buckl, C., Knoll, A.: A safety aware runtime environment for adaptive automotive control systems. In: Embedded real-time software and systems (ERTS2) (2014)

15. Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. In: Proceedings of the first workshop on Self-healing systems. pp. 27–32. ACM (2002)
16. Group, X.W.: Ieee standard for extensible event stream (xes) for achieving interoperability in event logs and event streams. IEEE Std 1849-2016 pp. 1–50 (2016). https://doi.org/10.1109/IEEESTD.2016.7740858
17. Gu, T., Sun, C., Ma, X., Lü, J., Su, Z.: Automatic runtime recovery via error handler synthesis. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 684–695. IEEE (2016)
18. Guo, C.G., Zhu, J., Li, X.L.: A generic software monitoring model and feature analysis. Journal of Software **6**(3), 395–403 (Mar 2011)
19. Illarramendi, M., Etxeberria, L., Elkorobarrutia, X., Sagardui, G.: Runtime observable and adaptable uml state machines: models@ run. time approach. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. pp. 1818–1827. ACM (2019)
20. Illarramendi, M., Etxeberria, L., Larrinaga, F., Sagardui, G.: Cresco framework and checker: Automatic generation of reflective uml state machine's c++ code and checker. In: 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). pp. 25–30. IEEE (2020)
21. Industriees, B.: Devops. https://www.bigindustries.be/blog/6-proven-steps-to-become-a-devops-engineer (December 2021), accessed on 2023-05-10
22. Jabbari, R., bin Ali, N., Petersen, K., Tanveer, B.: What is devops? a systematic mapping study on definitions and practices. In: Proceedings of the Scientific Workshop Proceedings of XP2016. pp. 1–11 (2016)
23. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Formal methods in system design **24**(2), 129–155 (2004)
24. Laboratory, F.S.: Monitoring oriented programming, http://fsl.cs.illinois.edu/index.php/MOP
25. Leemans, S.J., Fahland, D., Van der Aalst, W.M.: Scalable process discovery and conformance checking. Software & Systems Modeling **17**(2), 599–631 (2018)
26. Long, F., Sidiroglou-Douskos, S., Rinard, M.: Automatic runtime error repair and containment via recovery shepherding. ACM SIGPLAN Notices **49**(6), 227–238 (2014)
27. Mitsch, S., Platzer, A.: Modelplex: Verified runtime validation of verified cyber-physical system models. Formal Methods in System Design **49**(1-2), 33–74 (2016)
28. Pezzé, M., Wuttke, J.: Model-driven generation of runtime checks for system properties. Int. J. Softw. Tools Technol. Transf. **18**(1), 1–19 (Feb 2016). https://doi.org/10.1007/s10009-014-0325-2, http://dx.doi.org/10.1007/s10009-014-0325-2
29. Saxonica: Developers of the saxon processor for xslt, xquery, and xml schema, including the only xslt 3.0 conformant toolset (2022)
30. Schmidt, D.C.: Model-driven engineering. Computer-IEEE Computer Society- **39**(2), 25 (2006)
31. Schneider, D., Trapp, M.: A safety engineering framework for open adaptive systems. In: Self-Adaptive and Self-Organizing Systems (SASO), 2011 Fifth IEEE International Conference on. pp. 89–98. IEEE (2011)
32. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: Aspectc++: an aspect-oriented extension to the c++ programming language. In: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications. pp. 53–60. Australian Computer Society, Inc. (2002)

33. Van Der Aalst, W.: Process mining. Communications of the ACM **55**(8), 76–83 (2012)