Towards Continuous Integration in Model-Based Engineering of Automated Production Systems

Jakob Mund, Iman Badr, Safa Bougouffa, Birgit Vogel-Heuser

TUM-I1765
Towards Continuous Integration in Model-Based Engineering of Automated Production Systems

Abstract:
Continuous integration (CI) is widely used in software engineering. The observed benefits include reduced efforts for system integration, which is particularly appealing for engineering automated production systems (aPS) due to the different disciplines involved. Yet, while many individual quality assurance means for aPS have been proposed, their adequacy for and systematic use in CI remains unclear. In this article, we provide two key contributions: First, we propose a quality model for model-based aPS engineering approaches. Based thereon, we discuss the suitability of verification techniques for aPS and show their systematic integration in a CI process. As a result, we provide a blueprint to be further studied in practice, and a research agenda for quality assurance of aPS.

Keywords: Systems Engineering, Verification, Model-Checking, Simulation

1 Introduction

In software engineering, continuous integration (CI) refers to a set of principles and techniques that enable integrating changes made to the software system frequently, at least daily [1,2]. Together, they improve communication, developer productivity, quality of the developed system and project predictability (see, e.g., [3–5]). While approaches to implement CI may vary [6], they have in common to be based on a versioning system (e.g., SVN) as well as to automate the build and often parts of quality assurance. Reporting the result to developers in a timely manner with the intention to prioritize fixing the issue, CI aims to avoid what is sometimes referred to as ‘integration hell’.

During the development of automated production systems (aPS), integration was recognized as a key challenge due to the involvement of several disciplines, mechanical, electrical and electronic, as well as software [7,8]. aPS are complex and challenging systems due to their long life cycle, which requires adaptation of changing customer’s requirements (such as additional system configuration and self-healing capabilities) [9]. CI is currently used as part of the aPS’ development activities to control and improve the process of software creation [10]. However, adopting CI approach into the development process of aPS will require the integration of multi-disciplinary design process. Moreover, several approaches for quality assurance of discipline-specific and, but to a lesser extent, cross-discipline artifacts have been proposed [7,8,11–14]. Yet, their adequacy for and systematic use in a CI approach for aPS remains unclear.

In this article, we provide two key contributions. First, we provide a model of quality issues in engineering artifacts of a model-based engineering approach specifically developed for aPS [15]. Based on this model, we discuss selected state-of-the-art quality assurance means with regards to their suitability for CI of aPS and describe a basic CI process for developing aPS. This further covers anticipated variation points requiring a project-specific customization, e.g., how faults are handled. As a result of those contributions, we provide a solution proposal to be studied in practice, and a research agenda regarding quality assurance for CI of aPS.

The remainder of this paper is structured as follows: First, we outline the fundamental model-based engineering approach in Sec. 2 before defining the quality of such models by means of a quality model in Sec. 3. Next, in Sec. 4, we discuss complementary verification procedures, which we then integrate in the CI process presented in Sec. 5. Finally, in Sec. 6, we critically discuss current barriers to our approach and outline future research.

2 Model-Based Engineering of Automated Production Systems

Model-based engineering, i.e., the use of models as primary development artifacts, promises to cope with the growing complexity of mechatronic systems in general [7,16] and aPS in particular [17]. In this section, we outline a model-
The fundamental system model is based on Focus [18] to provide a strict formal semantics. Central to Focus is the notion of components and their interfaces. Firstly, a component’s static interface, i.e., its typed input and output ports, defines what messages the component may receive respectively send, and hence can be used to ensure structural compatibility between composed components. Secondly, the behavior observable at a component’s interface regarding those ports is called semantic interface. It is defined in terms of behavioral functions that map input streams to output streams. Intuitively, a stream is a sequence of messages sent or received over time on an input or output port, respectively. Commonly regarded as suitable for describing the system’s behavior [19], we use I/O state machines (e.g., Mealy machines) to specify behavioral functions operationally. Finally, individual components are connected by input and output ports via channels, provided the respective syntactic interfaces are compatible. While originally Focus was primarily conceived for modeling distributed embedded systems based on a discrete time execution, we rely on two recent extensions to model aPS holistically. Firstly, to also model e.g., mechanical aspects in a common language, the Focus theory was extended to support continuous time elaboration and data types [20]. Components that have discrete as well as continuous interfaces are referred to as hybrid components, and their behavior is defined by a modified version of the hybrid automaton called I/O hybrid state machine. In addition, we use the probabilistic extensions developed in [21] to model fault behavior [22]. Therefore, we generalize behavior functions to map input streams to a set of probabilistic spaces of output streams. Intuitively, it refers to a set of possible outputs and their associated probability. As input streams are mapped to sets of probabilistic spaces, the behavior specifications can be both non-deterministic as well as probabilistic. To specify the components’ behavior functions, we extend the state machine transitions with probability values.

Based on the formal system model, we now describe how we use it to holistically model aPS. aPS models consist of three related viewpoints on aPS, namely the requirements, process and system viewpoint [15].

In this paper, we focus on the system viewpoint which models the actual implementation of the aPS. Therefore, it models the resulting externally observable aPS behavior, i.e., the material flow, by means of specifying the system’s components and their composition, i.e., the architecture. The system viewpoint is further decomposed into three scopes, described in the following. The software scope provides a workspace for software engineers. It contains a model of the software architecture including the component hierarchy and their behavior. The syntactic interfaces of software components are purely concerned with data flow, and the behavior of software components is specified using discrete state machines. Next, the automation platform scope provides an encapsulated workspace for electrical/electronic engineers. It models the automation hardware from the sensor/actuator interface to the programmable controller that interfaces with the software scope. Typically, the platform scope models (potentially unreliable) physical communication channels including effects on the data flow. Finally, the mechanical context provides an encapsulated workspace for describing information from mechanical engineering. The interface of the context scope therefore comprises handover positions of workpieces as well as sensor and actuator connections to the platform scope, and behavior is described using hybrid and/or probabilistic components, or discrete abstractions thereof [20, 23].

The level of abstraction on which the aPS, in particular the platform and context scope, is modeled depends on the scope of the intended verification task: The deeper the scope of investigation, the more detailed are the formal models and the more complex are the verification tasks. For examples on the level of abstraction used, see [15, 22–24].

3 Quality of aPS Models

In this section, we introduce a basic model of quality issues that allows us to better understand the models’ characteristics that are subject to quality assurance. Note the crucial difference between the model presented here from quality models of systems, e.g., as described by the ISO/IEC Standard 9126 [25]. While the latter refers to attributes that characterize what a good system is, the former refers to attributes that characterize what good models for engineering such systems are. Obviously, there is a crucial but often implicit relationship between those models: attributes of the models created and used during engineering impact its outcome, i.e., the quality of the
actual system produced. In this article, we focus on model quality, and provide rationale in terms of the impact on the developed system whenever it may be unclear.

The model shown in Fig. 2 extends the fundamental models by Lindland et al. [26] and Unhelkar [27]. At the top level, we distinguish between syntactic, semantic and pragmatic quality. Subsequently, we will discuss syntactic and semantic quality in more detail, since those are most relevant to our application for automated quality assurance within CI, and postpone model aesthetics to our discussion in Sec. 6. For selected quality attributes, we provide exemplary quality issues relevant for interdisciplinary models of aPS according to our experience modelling these systems. While those quality issues are neither complete (i.e., they do not exhaustively define a quality attribute) nor necessarily sound for any engineering context in general (i.e., they might not impact the system’s quality in a particular situation), they provide a good understanding of the quality issues of aPS’ models.

### 3.1 Syntactic Quality

Syntactic quality refers to syntactic properties of one or several model elements. Since we use a modeling tool, our quality model neglects syntactic correctness [26] as the tool already prohibits using syntactically incorrect model elements. Therefore, we only distinguish between syntactic completeness and consistency as described in [28].

**Syntactic completeness** demands that no mandatory information is syntactically missing. Corresponding issues are mandatory attribute or child element of a model element that are not specified. For example, mandatory attributes are the name of a component or the type of a data port. The component name can be an arbitrary non-empty character sequence, while the data port type can be, i.a., a boolean value, a number, or a set (enumeration). Accordingly, a missing child element refers to, for instance, a component having neither sub-components nor a state-machine specifying its behavior. Similar mandatory attributes and child elements exist for many other model elements.

Complementary, **syntactic consistency** demands that the model is free of syntactic contradictions. Quality issues occur when a channel connects a source with a target port for which the permissions or types do not match. For example, an energy port cannot be connected to a data port or a material port, and vice versa. Furthermore, a boolean data port cannot be connected to a number data port or a set-valued data port. In addition, the input ports are only allowed to connect to output ports, and some ports may be hidden as part of a composition.

### 3.2 Semantic Quality

In contrast to syntactic quality, semantic quality is concerned with the meaning of the model. In case of our modeling approach, this primarily refers to the specified behavior of the aPS. We distinguish between **semantic correctness**, **completeness**, and **conformance**. While the former two characterize the relationship between the modeled aPS and the empirical (actual) aPS, more precisely, that the model is a correct and complete representation of the actual system, semantic conformance refers to whether the system conforms to the requirements imposed on the system. For this notion of conformance, we further distinguish between **intrinsic** and **extrinsic semantic** quality. Since they are the main focus in the remainder of this paper, we describe them in detail below.

**Intrinsic semantic conformance**: Intrinsic conformance are assertions about the modeled behavior that are not specific for a particular aPS model, but universally desired for practically almost any, or at least a wide range of, aPS. Therefore, intrinsic quality may be evaluated without an additional reference specifying the particular requirements for a specific aPS. From our experience of modeling
automation systems with our modeling approach, we are aware of the following issues:

Non-determinism: This issue refers to a behavioral state for which two successor states are reachable under the exact same conditions and inputs. While a non-deterministic specification may sometimes be intended, it often also happens by mistake, especially when modeling software behavior. In any case, non-determinism can have severe consequences, such as limited compositionality (cf. [29] for the merge-anomaly problem).

Multiple simultaneous writes: This issue refers to the phenomenon that an output port is written more than once in the same logical time step. For example, a transition of a state machine may contain two actions writing to the same port, or two concurrently active transitions might write to the same port at the same logical time. In such cases, the port valuation cannot be decided uniquely in general.

Unrealizability: The unrealizable issue occurs when (parts of) the model are infeasible to realize for any practical system. This may have various reasons, for instance, due to a lack of causality in the model, i.e., that a component’s output is specified to depend on inputs it did not receive yet, or that the specified behavior does contradict laws of nature, e.g., when a workpiece is specified to be moved faster than mechanically possible.

Workpiece Dead- and Lifelocks: This issue refers to unintended stagnation during processing of workpieces. More specifically, a deadlock occurs when a material is stuck during operation and cannot be processed further, e.g., because of incompatible handover between components. In contrast, a lifelock occurs when a material is continuously processed without ever being completed. For instance, this potentially occurs in plants with sorting capabilities, for which a wrongly specified algorithm may result in workpieces never reaching a destination position.

Lost Workpiece: The lost workpiece issue describes a state of the system in which a workpiece is located at a position that is unintended. Such a position may be reached either in case of hazards, e.g., as a result of failure of certain components, or as a result of a flawed specification which must be modified to reflect the intended system behavior.

Part collision: This issue describes the case that two mechanical parts of different physical components collide according to the model’s semantics. Avoiding part collisions is one of the most critical tasks during preliminary design of manufacturing systems, and must be avoided to prevent severe damage and maintenance cost during system operation.

Extrinsic semantic conformance: Extrinsic conformance is concerned with the relationship between the modeled system and the requirements that are specific for the individual aPS under development. For example, a constraint might limit the maximum energy flow over an energy port or the maximum duration of a workpiece might take to be processed by the aPS. Note that unlike intrinsic semantic conformance, a reference to decide extrinsic semantic conformance is required, e.g., the requirements specification.

4 CI Verification Techniques

In this section, we present techniques suitable to verify some of the model qualities presented in the previous section, and which can be applied to the modeling approach presented in Sec. 3 as part of CI. To this end, we first describe characteristics that qualify verification techniques for use in CI of aPS.

4.1 Requirements

For a verification technique to be suitable for CI, we argue that it must possess the following characteristics:

1.Artifact Applicability: It must apply to the available engineering artifacts, in our case, the models or artifacts automatically derived therefrom (e.g., source code or...
In addition, any artifacts required for the verification procedure, such as formalized requirements or test-cases, must also be readily available or obtainable from activities consistent with the aPS engineering methodology.

2. Quality Relevant: For the verification procedure to be effective, it must reveal quality issues that are relevant for aPS. The quality model presented in Sec. 3 provides an overview of the relevant model qualities, and the quality issues presented describe selected exemplary model properties to be discovered by the verification procedures.

3. Actionable Feedback: It must provide feedback that benefits the engineers in locating and fixing issues, e.g., by explaining and pinpointing to the root cause. Also, (part of) the feedback must be understandable (e.g., using a parser) by the CI server software in order to determine if the verification step was successful or not, and hence, shall be passed to the next stage.

4. Prior to Deployment: The build and test stages of CI are prior to the software being actually deployed on the productive aPS. Therefore, all verification procedures are limited to development (including preproduction) environments. If the verification procedure requires historic field data, the verification approach is only applicable in iterative development approaches for the latter iterations.

5. Performance: Since developers are expected to integrate their changes several times, but at least once, per day, the verification performance must reflect this demand. Hence, verification procedures that take longer than seconds or a few minutes are inadequate if invoked for each commit, respectively a few hours if part of a nightly build.

6. Automated: In CI, the burden of performing verification is transferred from the engineers to the CI server. This, of course, requires the procedures to be automated. As mentioned before, this includes invocation of the procedures and evaluating their result. In case the verification requires additional artifacts based on manual steps, such as providing a specification or test-cases to verify against, this must be accomplished beforehand and be consistent with the engineering approach, which is achieved by following a test-driven methodology [30].

Consequently, applicability of verification procedures for CI is a function of the engineering context, e.g., the engineering process, tools used or the size of the system.

4.2 Suitable Verification Techniques

Based on the requirements presented in the previous section, we now provide an overview of quality procedures which can be applied for CI and our modeling approach, and discuss the their prerequisites and validity. To this end, we discuss techniques of testing, co-simulation, and model-checking in the context of aPS verification, as illustrated in Fig. 3.

**Fig. 3:** Classes of verification techniques applicable for CI of aPS

**Testing:** Testing-based approaches manipulate the executing system with certain, predefined stimuli, observe the corresponding output, and compare it to reference values. Unfortunately, as test-cases cannot be executed on the actual plant as part of CI (see Sec. 4.1), testing-based approaches cannot verify the software achieves the desired aPS behavior. Yet, since our approach allows us to automatically synthesize executable code from the model, it can be used for software testing. For software testing, we can use black-box, white-box and model-based testing approaches, and for unit as well as integration tests (see, e.g., [31]). As a hybrid form between testing and simulation, virtual commissioning (or hardware-in-the-loop) gained widespread interest for aPS [12]. Essentially, this technique combines executable software deployed on actual automation hardware with simulated behavior of the mechanical context to obtain a semi-simulated plant behavior model. Besides the simulation model and testing information, this technique also requires an automated deployment of the software and an automated hardware and simulation setup in which, e.g., configuration parameters such as PLC clock speeds can be updated by the CI server.
Co-Simulation: Simulation refers to the virtual (symbolic) execution of the system by means of a simulation model, again with predefined stimuli and output expectations. As testing is generally preferred to simulation for software alone, we only consider co-simulation techniques that simulate, besides software, also the behavior of the automation hardware and/or (parts of) the mechanical context. Simulation requires an executable modeling language, an execution engine, a specification of input stimuli, and an oracle of expected outputs. Since in our approach, the software, automation platform and mechanical context are all modeled using the FOCUS modeling theory, we indeed have such an executable aPS behavior specification. A frequently raised concern about simulation is performance, in particular, when used for systems with continuous behavior. To overcome this limitation, we rely on two techniques. First, during modeling, we use discrete abstractions of continuous behavior, as presented in [23]. Second, during simulation, we rely on sampling techniques that adjust the sampling rate according to regions of interest to improve precision and performance [20]. The simulation engine itself is implemented in AF3, the CASE tool also used for modeling. Finally, we proposed two possibilities for specifying input stimuli and an oracle of expected outputs. On the one hand, system-level requirements can be formalized by means of sequence charts [32] or assertions expressed in temporal logic [15]. Alternatively, in [24], we suggested a formal specification suited to specify the technical process to be performed on the plant, and against which the system can be verified.

Finally, in this section, we describe how the presented verification techniques can be integrated in a CI approach, including anticipated variations point, in the context of aPS engineering.

5 CI Process Proposal

Model-Checking: Model-checking refers to the exhaustive verification of a system model against its formal specification. In contrast to testing and simulation, model-checking does not require a specification of inputs, but instead, relies on assumptions about the environment. Essentially, for model-checking, we translate our interdisciplinary models into representations amenable to state-of-the-art model checker, such as NuSMV, HyTech, or PRISM, using model-to-model transformations. Based on the same abstraction and specification techniques used as described for simulation, we were able to prove crucial properties of aPS systems [15, 22, 24]. However, due to its exhaustive nature, model-checking suffers from scalability issues, with run-times of minutes up to several hours.

5.1 Basic Process Model

The essential CI work-flow for our model-based aPS engineering approach is exemplary illustrated in Fig. 4. We assume the aPS model is stored in a central repository. Each involved engineer works on a local copy of this model, and commits his or her changes at least on a daily basis. Within a commit, the changes in the local copy are merged with the baseline model, with the engineer being responsible for resolving potential merge conflicts. We also suppose basic syntactic checks to ensure syntactic completeness and consistency (see Sec. 3.1) are performed prior to or during merging. Each commit is identified by the CI server (e.g., JENKINS), which is then responsible to initiate and monitor the built process. If the built fails, the engineer who carried out the commit is notified. Otherwise, if a successful built is obtained, the CI server coordinates and monitors verification. In the following, we describe this phase in more detail. We propose to organize the verification techniques as shown in Fig. 5. Essentially, we break the verification down into two distinct work-flows, namely (1) the rapid techniques conducted for each commit, and (2) the system-level model-checking conducted once a day, e.g., during a nightly built.

The (1) rapid verification is organized along an increasing verification scope, starting with verifying the software in isolation. Therefore, unit tests are executed first. If all unit tests pass, integration tests of several modules or the control software as a whole are executed. In case the additional efforts for specifying logical constraints at the software-level are acceptable, exhaustive model-checking of the control software can be done in parallel to testing, and should be feasible within seconds or a few minutes due to the control software’s limited structural complexity. Any failed test-cases or flaws revealed by model-checking are immediately notified to the software engineer responsible for the commit. After the software system is verified at the software scope, we use co-simulation to verify it also has the desired effects on the automation hardware, and ultimately, achieves the desired plant behavior. Therefore, the next step is to verify the integrated hardware/software system behaves according to constraints. Since providing such constraints imposes an additional burden to engineers, we consider this step to be optional depending on its cost-effectiveness on a case-by-base basis. Finally, our
continuous verification procedure suggests to perform a co-simulation against formalized requirements, a formal specification of the technical process, or both (see Sec. 4.2). In case any co-simulation reveals a flaw, the engineer responsible for the commit is notified, which includes, e.g., electrical or mechanical engineers. This system-level co-simulation completes the verification procedures invoked for each commit. In case an automated hardware setup exists, this step can also be replaced by HIL verification.

In addition, our continuous verification approach also includes exhaustive but computationally intensive (2) system-level model-checking. To this end, we suggest to model-check the aPS model once a day, preferably during a nightly build, and against the requirements specification and/or formal process specification. In case a flaw is revealed, all engineers who committed a change during the day are notified, and share the responsibility to resolve this interdisciplinary integration issue.

If all verification steps succeed, we consider the software ready for deployment and acceptance testing. Depending on the concrete hardware used and the quality management policies, this step may also be automated as part of a continuous deployment process as a long-term vision.

5.2 Variation Points

Based on the classification by Ståhl and Bosch [6], we now discuss selected variations points to be considered when applying the presented CI approach in a specific project.

Fault handling: In the CI process described above, we implicitly adopted a strict position in handling faults, with the engineer(s) breaking the current version in the central repository being responsible to immediately fix the issue, or reverting the changes to restore a working current version. In some cases, a more relaxed handling of faults, in which certain faults are permitted to persist over some period of time, may be more adequate. For instance, allowing integration flaws detected by co-simulation and model-checking at the aPS level to persist during early development of new features allows engineers to focus on extending the models first. However, we would advise to limit this relaxation to novel system features only, e.g., by insisting completed features are not harmed, and only for a limited amount of time, e.g., a sprint. Otherwise, the benefits of CI severely diminish.

Testing new functionality: While verification as part of CI must ensure so far correctly implemented functionality is not harmed by a change, we envision it to also be used to verify newly created features are verified. Besides potentially requiring a more relaxed policy in handling
faul ubiquitous, as described in the previous paragraph, this also makes a test- or specification-driven engineering approach necessary, since the test cases or formalized requirements have to be created up front when starting to implement new features.

**Pre-integration procedures:** The procedures that are executed on the engineers’ local development environments constitutes another variation point. For our approach, we suggest to at least check for syntactic issues (Sec. 3.1) locally and resolving any occurring issues before committing any changes to the central repository. In most cases, we would also advise software engineers to run unit tests on their local machines. In contrast to this advise, some researchers have argued that delegating this task to the central repository allows to free up more time from the engineers, however, at the expanse of potentially breaking the build more often.

**Integration on broken builds:** Last but not least, implementing a CI process also requires to decide how to cope with changes in case of a broken build, e.g., because the model cannot be compiled or verification revealed flaws. Here, the central question is whether it is appropriate to allow integration of changes into the repository that are not fixes to the revealed problems, or not. According to [4], the consensus among researchers is that such commits can be problematic, in the sense that it may greatly complicate the problems for the people engages in fixing the build, but whether enough so to actively try to prevent them, is contended.

### 5.3 Technical Implementation

To evaluate technical feasibility, we outline a tool infrastructure that can be used to exemplary implement the described CI process. At the client side, the AF 3 CASE tool\(^1\) can be used to model aPS [15], which are saved using an .AF3 XML representation. Any change done and committed by an engineer is then merged into a model-repository, e.g., using a text-based (e.g., using SVN, GIT\(^2\)) or, favorably, model-based versioning system such as CDO\(^3\) or AMOR\(^4\). If the CI approach should also verify extrinsic semantic correctness of novel functionality, the repository must also include specifications and/or test-cases. At the server side, a CI server such as JENKINS\(^5\) is responsible to initiate and monitor the build and verification steps of the workflow. To this end, it must be made aware of any successful commit to the common model repository, e.g., by actively polling or hooking into the merge routine. For each build, Jenkins must then invoke AF 3’s code generation engine for the software scope of the aPS model, pass the results to a compiler/linker (e.g., GCC\(^6\)), and monitor if those steps are successful. In case they are, the verification procedures (Fig. 5) are to be invoked next. For model-checking, we must first apply a model-to-model transformation from the a3 aPS model to the model understood by the model checker, depending on the property to be verified. Currently, we support automated translations to Kripke structures (.smv) for NuXmv\(^7\) and Markov decision processes (.pm) for PRISM\(^8\) for qualitative and probabilistic specifications, respectively. In a second step, this model is, together with the properties to be verified, passed to those model-checkers. Like most model-checkers, they offer batch/command line interfaces, and can thus be automatically invoked and monitored by Jenkins. For simulation, this step reduces to invoking AF 3’s simulation engine. However, in contrast to model checking, a simulation case must also be provided, which specifies the inputs for the aPS system, a simulation bound (i.e., the number of steps), and a strategy how to proceed with non-deterministic transitions (e.g., choose randomly or explore all). In case an error occurs during any of the steps during build or verification, Jenkins reports the results back to the engineers, e.g., using the MAILER plug-in.

### 6 Conclusion

In this article, we described a model-based engineering approach for aPS based on inter-disciplinary modeling using a formal modeling language. For such models, we provided an overview of quality characteristics as well as exemplary concrete issues, and described how state-of-the-art verification techniques can address the models’ conformance to the system requirements. Finally, we described how those verification techniques can be integrated within a CI process for aPS and discussed anticipated variations. However, we recognize that the wide applicability of CI in practice still requires to overcome a number of challenges in the

---

1. [https://AF3.fortiss.org/](https://AF3.fortiss.org/)
3. [https://www.eclipse.org/cdo/](https://www.eclipse.org/cdo/)
5. [https://jenkins.io/](https://jenkins.io/)
7. [https://nuxmv.fbk.eu/](https://nuxmv.fbk.eu/)
future. Subsequently, we outline what we consider key challenges, as well as the research opportunities identified from our work.

Barriers in Practice: We see the need for reducing the efforts associated with obtaining the inter-disciplinary model and necessary verification information for our approach. To this end, instead of modeling the aPS’ hardware and mechanics by hand, we envision (parts of) those models to be automatically derivable from other models more widely available, e.g., CAD models. Also, we suspect the formal specifications originating from theoretical computer science and which we rely on for verification, e.g., linear temporal logic, are not well suited for aPS engineering, and require more accessible notations and specification techniques.

The tool support for the techniques outlined in this paper in combination with the recent advances in related fields, such as model versioning, allowed to come a significant step closer to accomplish an automated CI aPS engineering approach. This is in particular true to obtain a minimal viable product that does, for instance, not include model-based versioning nor advanced reporting. In this case, extensions are mostly necessary to the AF 3 interface, i.e., support command-line interfaces for back-end functionality (model-transformation and code generation, basically). The next step would be to make the engineering experience more seamless. This would include developing robust connectors, primarily for AF 3, CDO/AMOR, and JENKINS, e.g., in the form of plug-ins. While requiring considerable efforts, we do not consider those particularly challenging from a scientific point of view. However, for seamless simulation-based verification, some more fundamental research is necessary, in particular, for generating aPS simulation inputs efficiently and effectively.

Future Research: Our work motivates a number of questions for future research. The reader may have noticed that our CI approach does not completely cover all quality characteristics described in Sec. 3. Indeed, it remains unclear if and to what extent the models’ aesthetics, which become important when the model is not to be understood by a program but a different human reader, can and should be considered, similar to what coding style checks aim to achieve for source code. Moreover, our verification approach focused on conformance of the model to requirements, while neglecting whether the model is indeed a complete and sound representation of the physical aPS. Those questions are, however, crucial for deciding if continuous deployment is reasonably acceptable, not even accounting for the technical challenges associated with it, e.g., automating deployment for automation hardware.

Acknowledgment: This work was supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design for Future – Managed Software Evolution.

References

REFERENCES


