AUTOMATED VERIFICATION OF CARDIOVASCULAR MODELS WITH CONTINUOUS INTEGRATION TOOLS

M. Bachler[(a)], B. Hametner[(b)], C. Mayer[(c)], J. Kropf[(d)], M. Gira[(e)], S. Wassertheurer[(f)]

[(a) - (f)]AIT Austrian Institute of Technology GmbH, Health & Environment Department, Biomedical Systems
[(a) - (b)]Vienna University of Technology, Institute for Analysis and Scientific Computing

[(a)]martin.bachler@student.tuwien.ac.at, [(b)]bernhard.hametner@ait.ac.at, [(c)]christopher.mayer@ait.ac.at, [(d)]johannes.kropf@ait.ac.at, [(e)]matthias.gira@ait.ac.at, [(f)]siegfried.wassertheurer@ait.ac.at

ABSTRACT
Models in general, but especially in medicine, need extensive testing and verification to ensure that they do not contain errors and produce correct results. Traditionally, this happens after completing the development. In this work an approach to automated and continuous testing, verification and documentation based on a continuous integration tool is presented. This practice has several advantages in comparison to the traditional way of verification. As the model is verified after every single change that is made to it, one benefit is the earlier and more precise tracing of errors. Another advantage is the aggregation of code generation, software building, testing, verifying and documentation in one tool to ensure maximum automation and to reduce expenditure of time. Furthermore, due to the integration of central versioning systems, it makes working in development teams easier. In this work, the development processes of two cardiovascular models are incorporated into a continuous integration system.

Keywords: continuous integration, cardiovascular model, model verification, development tools

1. INTRODUCTION

In contrast to model checking, which ensures the formal validity of a software system, model verification tests the correctness of the results derived from a given model. In medicine, the verification of physiologic models can be done, for example, by comparing these results with real world measurements.

This verification process traditionally starts after the development of the model is completed and executable software has been build, whereas the building process itself is usually carried out manually and step by step. Finally, documentation of model and verification is produced by hand. All these steps have to be carried out each time the model is changed.

Figure 1 shows a diagram of this typical development approach, e.g. as described by Bachler et al. (2011).

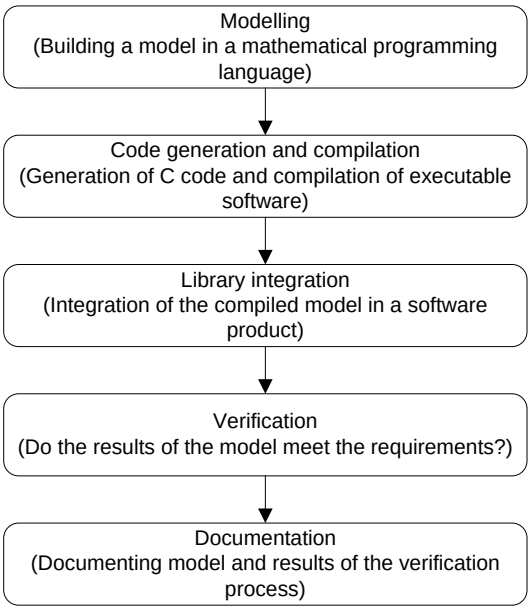

Figure 1: Typical Development Process of a Software Product based on a Model

Among others, one big disadvantage of this approach is the late verification of the model. Finding an error in the fourth step (verification) of the process usually forces the developer to go back to step two (code generation and compilation) or even step one (modelling), depending on the mistake.

Another drawback can be found in environments where the model is subject to continual changes. Usually, several changes are made to the model before the building process is started again. Therefore it is not easy to track back errors found during the verification and relate them to a specific change.

Although these issues are addressed by best practice paradigms such as “test early, test often” they tend to reoccur in many different software development processes, not only in model based algorithm development. Following these fundamental practices seems to be harder than one would expect. The automation of the entire building process can help to enhance the frequency of testing and verification (Duvall, Matyas, and Glover 2007).

In this work the automation of the development processes of two software products containing cardiovascular models using the continuous integration tool and open source software Jenkins in the version 1.473 is described.

2. METHODS AND MATERIALS

In the first part of this chapter, two concrete software products containing cardiovascular models are presented. Their development processes, which were transferred into a continuous integration system, will be described. One of the models is dealing with the calculation of certain parameters using pulse wave analysis, the other is used for the detection of features in electrocardiography.

The second part deals with continuous integration tools in general and the software tool Jenkins in particular.

2.1. Pulse Wave Analysis

Pulse wave analysis (PWA) in general deals with the determination of several cardiovascular parameters calculated from the pulse wave travelling through human arteries. The AIT Austrian Institute of Technology GmbH developed a non-invasive and easy to use method based on recordings of the pulse wave by means of an occlusive blood pressure cuff (Wassertheurer, Mayer, and Breitenecker 2008; Wassertheurer et al. 2010; Hametner 2011; Hametner et al. 2012; Weber et al. 2011; Wassertheurer, Hametner, and Weber 2011; Nunan et al. 2012).

Certain software algorithms for the calculation of parameters for the aortic blood pressure, arterial stiffness and wave reflection are part of this method. As this system is subject to ongoing research, the underlying models are changed continually. Furthermore, several developers are involved in the modelling and building process. Hence, these models and algorithms are perfect candidates to be incorporated in a continuous integration system.

The model based algorithms used for the pulse wave analysis are developed in the mathematical programming language MathWorks MATLAB® in the version R2007b and converted to C code using Embedded MATLAB®. This C code is compiled into dynamic link libraries using standard C compilers and integrated in a software product written in the programming language Java. The original development process followed the steps shown in Figure 1.

The whole development process is summarised in Figure 2. In the original setting, the transition from MATLAB® code to the DLL was semi-automatic using shell-scripts. The integration of the DLL in the Java-Environment, software tests and verification as well as the documentation was done manually.

2.2. Electrocardiography

Electrocardiography (ECG) is the measurement of the electric activity of the cardiac muscle. It is a non-invasive, painless technique and is widely used in the assessment of heart failures. The tracing of one heartbeat consists of a P wave representing the atrial depolarization, a QRS complex showing the ventricular depolarization and a T wave at the ventricular repolarisation.

The software algorithm is able to detect beginning, peak and end of the QRS complex, the P and the T wave of each heartbeat automatically and in real time (Bachler et al. 2011).

The results of this algorithm are verified by comparing them to annotations made by medical experts with data from different ECG databases (Goldberger et al. 2000).

Like the algorithms for pulse wave analysis, this software is written in MATLAB®, converted to C code, compiled to a dynamic link library and integrated in a software product written in Java (shown in Figure 2). Again, the transition from MATLAB® code to the DLL is semi-automatic, whereas integration, verification and documentation are done by hand. Therefore several build steps from the pulse wave analysis algorithm can be reused.

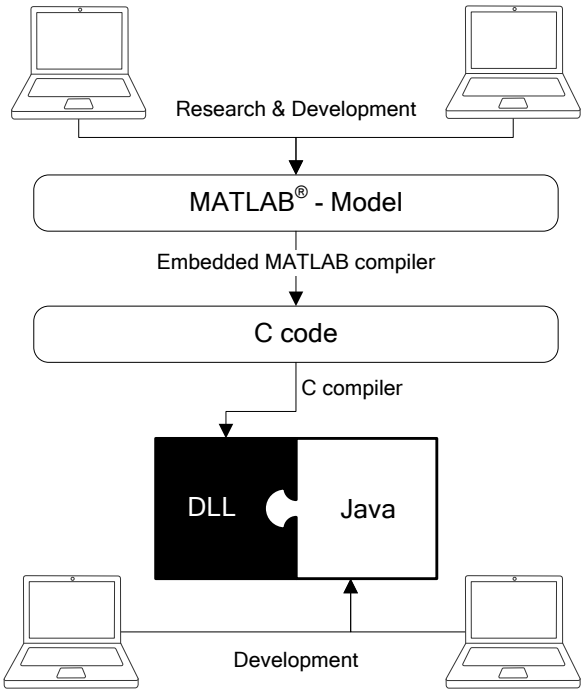

Figure 2: Development of different parts of the software system for Pulse Wave Analysis and ECG Analysis

2.3. Continuous Integration

Continuous integration helps implementing "best practices" in software development by automating the whole building process (code generation, compilation, testing and verification) and the documentation thereof (Duvall, Matyas, and Glover 2007).

Focused mainly on the principles of centralisation, "test early, test often", automation of build and documentation, and feedback, a continuous integration system usually features (as shown in Figure 3):

- A version control repository,
- A continuous integration server,
- Build scripts, and
- A feedback mechanism.

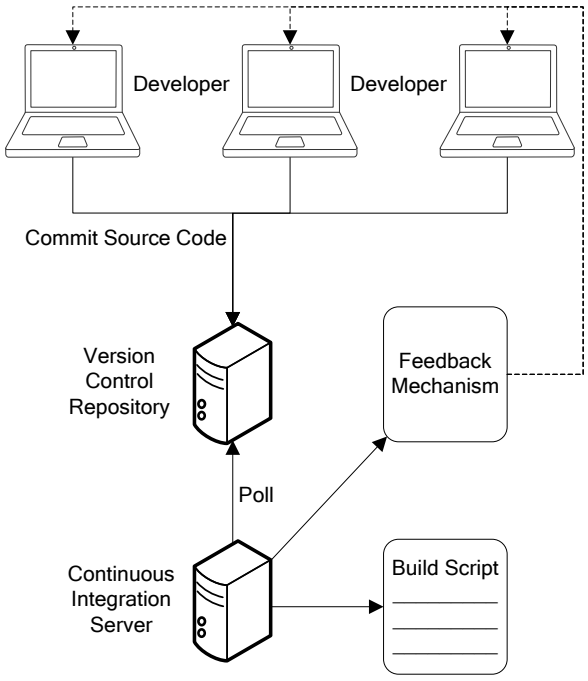

Figure 3: Basic Components of a Continuous Integration System (Duvall, Matyas, and Glover 2007)

2.3.1. Centralisation

In research and development teams, usually a file server is utilized as central data storage accessible to all team members. Typically a version control system such as Concurrent Versions System (CVS) or Apache Subversion (SVN) is used to keep track of all changes made to the files.

Continuous integration systems support the practice of using central versioning systems and integrate them seamlessly without any change necessary. A central continuous integration server frequently checks for changes in the source code stored in the version control system. If a change is detected, the building process is carried out according to some predefined build scripts.

2.3.2. Test Early, Test Often

Testing is part of the building process. Automated software tests (including the verification of the results obtained using the model) are executed each time after the software was compiled. As the building process is started after each change in the source code that is submitted to the version control system, the only thing developers need to do is to commit their code every time they add or change something. As a consequence, every single change leads to a full test and verification of the whole system and therefore also the model. This narrows down bug tracking to where did the error occur (i.e. in which development step) and when did it occur (i.e. after which change in the model), therefore making errors a lot easier to resolve (Duvall, Matyas, and Glover 2007).

2.3.3. Automation of Build and Documentation

With tools such as GNU Make (originating from UNIX systems and mainly used for the C programming language), Apache Ant and Apache Maven (primarily for development in Java), build automation tools are already widely used. But instead of running these tools on the machines of developers (which probably prevents them from doing something else in the meantime), they are incorporated in the continuous integration system and executed on a dedicated continuous integration server.

Therefore, existing build scripts can be reused easily in a continuous integration system. Furthermore, different sorts of scripts can be combined to create a fully automated build environment.

In addition to the compilation process and automated testing, sophisticated scripts allow the automated verification of the results of the model based algorithms described above and the automated generation of verification reports.

2.3.4. Feedback

As building, testing and documentation is completely taken over and automated by the continuous integration system, there has to be a mechanism to inform the developer of success or errors in the build. Usually, the continuous integration server is configured to send an e-mail to either a predefined address (probably the coordinator of the team) or to the developer that initiated the building process by committing code changes.

If errors occur during the building process, the feedback contains detailed descriptions of them to allow fast and easy debugging. Otherwise, build artifacts are generated. These usually consist of executable software or compiled libraries, test results and verification reports.

2.4. The Continuous Integration tool Jenkins

Jenkins is an extendable, web based continuous integration tool written in Java and published under the open source MIT license. It supports several build tools such as Apache Ant, different versioning systems such as Apache Subversion and automatic software testing tools. It is a fork (spin-off) from the continuous integration system Hudson supported by Oracle (Wiest 2010).

Originally, it was designed for Java projects only, but with the capability of using plugins its features can be extended far beyond this limited purpose (Wiest 2010).

Projects can be created and managed via a web interface using an ordinary browser. Therefore, every developer can access the same configuration data, adapt them or check the status of a certain project (Wiest 2010).

Jenkins is not limited to one central continuous integration server but can incorporate several distinctive nodes running different operating systems. This feature is especially essential if the source code has to be compiled for different platforms such as Linux, Mac OS X or Microsoft Windows, but can also help distributing the work load to several building machines (Wiest 2010).

2.5. Using Jenkins for the Development of the Algorithms based on Cardiovascular Models

As described earlier in this section, some parts of the development process were already automated using shell scripts. Also the version control system Apache Subversion in the version 1.6 was already in use. So, when implementing Jenkins with the projects for pulse wave analysis and ECG analysis, the main work was the combination of all single steps into one completely automated process. The biggest challenge was the creation of automated verification reports. In contrast to standard software tests, which primarily give a yes/no-answer to the question of the absence of runtime-bugs, verification has to quantify the difference between the results obtained using the model and a reference. Therefore, the report of the verification process cannot be simply the output of a standard software test but has to include extensive statistical analyses of the results.

2.5.1. Assessment of the Initial Situation

To create an overview of the steps necessary for porting the whole development process to Jenkins, an assessment of the initial situation has to be performed:

1. Modelling: The models used in pulse wave analysis and ECG analysis are written in the programming language MATLAB®. As this is the creative part of the development process done by researchers and developers, it is not possible to automate this task.
2. Code generation: Using scripts written in MATLAB®, models and algorithms from step 1 are converted to code in the C programming language.
3. Code compilation: In this step, dynamic libraries are created for Linux, Mac OS X and Microsoft Windows. Therefore, three building machines with different operating systems are in use. Shell scripts for these compilation processes already exist, but they have to be executed on each machine manually.
4. Library integration: The three libraries built in step 3 are integrated in a Java project, which again is build using a shell script on one of the machines mentioned in the step above.
5. Verification: The verification consists basically of three parts: verifying the MATLAB® model itself, verifying the libraries integrated in the Java project and running automated software tests. For the first part, a MATLAB® script is used to compare the results derived using the models with a reference and to quantify the differences. The libraries are verified manually on their respective operating system (Linux, Mac OS X and Microsoft Windows) by executing the Java software, loading the reference data, performing the calculations and exporting the results. These results are then compared and quantified using a MATLAB® script. The automated software tests are executed using a shell script and the testing framework TestNG (Beust and Suleiman 2007).
6. Documentation: The documentation of the results of the verification is done manually by summarising all results generated in step 5 and describing the changes since the last version of the software.

2.5.2. Adaption to a Fully Automated Continuous Integration

Several steps were taken to adapt the existing development procedures and to integrate them in Jenkins:

1. Jenkins was configured to access the version control repository and to frequently check for modifications of the model. If a modification is detected, it will perform a clean check-out of the source code and start the whole building process.
2. To automatically run MATLAB® scripts, MATLAB® was installed on the same machine as Jenkins. These scripts can be executed by Jenkins through a shell script which starts MATLAB® without user interface and runs the MATLAB® script. These scripts are used for code generation and verification.
3. To compile the C source code for different operating systems, three machines where set up to run Jenkins: one Linux, one Mac OS X and one Microsoft Windows machine. Also, three Jenkins projects were created, one for each platform. Each was configured to be built only on one designated machine using the appropriate shell scripts. The scripts could be reused without modification (Berg 2012).
4. The verification of the libraries was automated by transferring the evaluation of the reference data to the automated TestNG tests. Instead of loading the reference data and exporting the results manually, these steps have been added to the already existing TestNG tests. These tests are executed by Jenkins automatically after compilation is finished.
5. To automate the documentation of the verification as far as possible, the MATLAB® script quantifying the differences between model, libraries and reference was adapted to write these results into a file. A source file in the document markup language LaTeX was

prepared to automatically read and summarise these results. The LaTeX typesetting system is executed by Jenkins after all other development steps have been finished successfully to produce a PDF document containing all results of the verification process. This document also features the possibility of adding text to allow a manual description of the results. Therefore, the repetitive part of the generation of the documentation was automated using Jenkins. The creative part, which includes a detailed description of the changes in the model as well as an interpretation and discussion of the results of the verification, is still left to researchers and developers.

3. RESULTS

Figures 4 and 5 present a qualitative comparison of the workflow before and after the introduction of the continuous integration system to the development process of the software products containing the cardiovascular models.

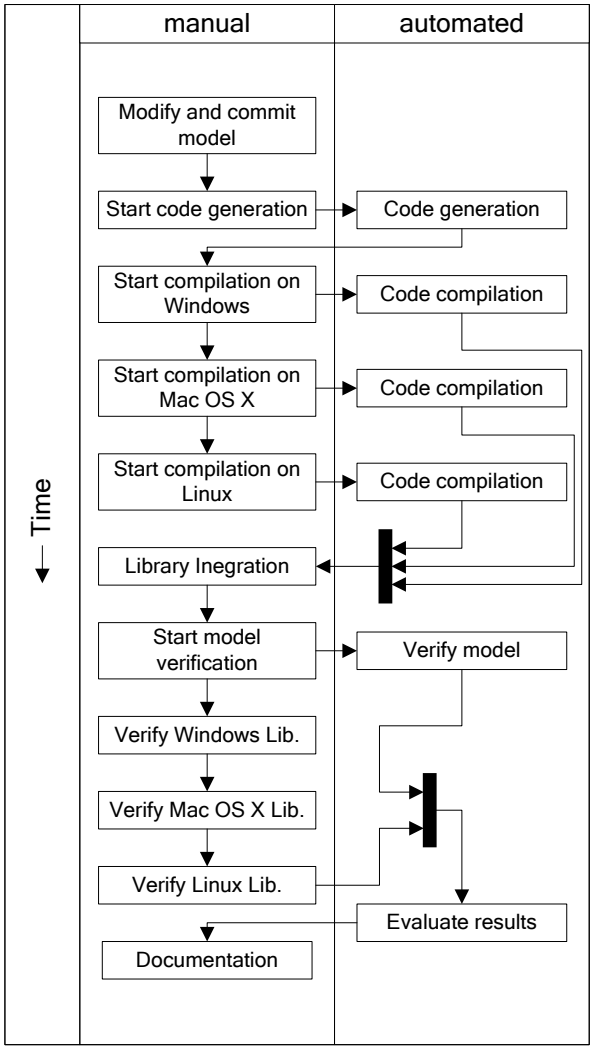

Figure 4: Manual and Automated Tasks in the Development Process without Continuous Integration

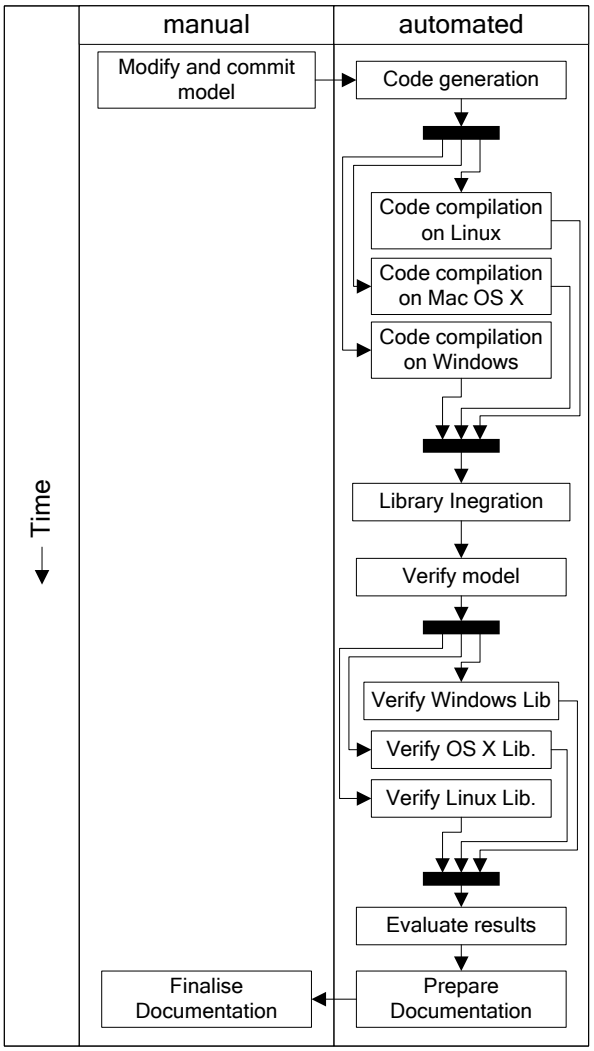

Figure 5: Manual and Automated Tasks in the Development Process with Continuous Integration

Table 1 gives an overview of the time spent by a single developer on different sorts of tasks, assuming that there are no errors in the model (please note that the given time spans are rough estimates and that the true values depend heavily on the amount of reference data used for verification and the time spent on the creative part of the documentation).

Table 1: Comparison of Time Spent by a single Developer on Different sorts of Tasks with and without a Continuous Integration System (CI)

Time spent by developer	Without CI	With CI
Overall	60 min	15 min
On repetitive tasks	45 min	0 min
On creative tasks	15 min	15 min

3.1. Discussion

The heavy overhead of repetitive tasks burdening the developer in a development process without a continuous integration system (see Table 1) usually lowers the frequency of code compilation, code testing

and model verification. Therefore, usually several changes are made to the model and only verified once. Assuming that one of these changes leads to an error during the verification process, it is hard to determine the source of the error as there are multiple possibilities. Shifting this overhead to the continuous integration system and triggering the whole build-and-verify-process after every single change that is made to the model leads to a higher frequency of builds and therefore a higher frequency of verifications. Errors are detected immediately and can be resolved in less time.

4. CONCLUSION

Using the continuous integration tool Jenkins and several of its extensions, the development, verification, and documentation processes of software systems containing cardiovascular models were automated. The developer is relieved of repetitive tasks and the frequency of model verifications during the development is raised. Lowering the expenditure of time of the building process due to automation and the time needed for the fixing of bugs because of earlier and more accurate error reports lead to a speed up of the release of new versions of the software.

REFERENCES

Bachler, M., Mayer, C., Hametner, B., and Wassertheurer, S., 2011. Automatic detection of QRS complex, P and T wave in the electrocardiogram. *Abstractband / 21. Symposium Simulationstechnik: ASIM 2011*, 37. September 7-9, 2011, Winterthur, Switzerland.

Berg, A, 2012. *Jenkins Continuous Integration Cookbook*. Birmingham:Packt Publishing Ltd.

Beust, C., and Suleiman, H., 2007. *Next generation java™ testing: testng and advanced concepts*. Boston:Addison-Wesley Professional.

Duvall P. M., Matyas S., Glover A., 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston:Addison-Wesley Professional

Goldberger, A. L. et al., 2000. PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals. *Circulation* 101:e215-e220.

Hametner, B., 2011. *Arterial Pulse Wave Analysis: Impact of Models for Impedance and Wave Reflection*. Doctoral thesis. Vienna University of Technology.

Hametner B., Weber T., Mayer C., Kropf J., Wassertheurer S., 2012. Effects of Different Blood Flow Models on the Determination of Arterial Characteristic Impedance. *Preprints MATHMOD 2012 Vienna – Abstract Volume*. 2012:262.

Nunan, D., Wassertheurer, S., Lasserson, D., Hametner, B., Fleming, S., Ward, A., and Heneghan, C., 2012. Assessment of central haemomodynamics from a brachial cuff in a community setting. *BMC Cardiovascular Disorders* 12:48.

Wassertheurer S., Mayer C., Breitenecker F., 2008. Modeling arterial and left ventricular coupling for non-invasive measurements. *Simulation Modelling Practice and Theory* 16:988-997.

Wassertheurer S., Kropf J., van der Giet M., Baulmann J., Ammer M., Hametner B., Mayer C., Eber B., Magometschnigg D., 2010. A new oscillometric method for pulse wave analysis: comparison with a common tonometric method. *Journal of Human Hypertension* 24:498-504.

Wassertheurer S., Hametner B., and Weber T., 2011. Model based estimation of aortic pulse wave velocity. *Artery Research* 5:162.

Weber, T., Wassertheurer, S., Rammer, M., Maurer, E., Hametner, B., Mayer, C.C., Kropf, J., and Eber, B., 2011. Validation of a brachial cuff-based method for estimating central systolic blood pressure. *Hypertension* 58:825-832.

Wiest, S., 2010. *Continuous Integration mit Hudson/Jenkins: Grundlagen und Praxiswissen für Einsteiger und Umsteiger*. Heidelberg:dpunkt Verlag.