A Model-based Approach for Build Avoidance

Milena Neumann
PTV Group
Karlsruhe, Germany
milena.neumann@ptvgroup.com

Kiana Busch
Karlsruhe Institute of Technology
Karlsruhe, Germany
kiana.busch@kit.edu

Robert Heinrich
Karlsruhe Institute of Technology
Karlsruhe, Germany
robert.heinrich@kit.edu

***Abstract*—In large software systems, we frequently encounter change scenarios which require long build times. In many cases, it would suffice to build only a subset of the dependent build components to generate sound build results. Current approaches for change-specific identification of affected build components rely on knowledge about the language-specific propagation of changes, which renders them inapplicable to multi-language systems. In this paper, we present a model-based approach to derive the affected build components for a change scenario using an existing change propagation approach. This way, we make the advantages of a set of change-specific dependencies also accessible to those members of the development team who are less knowledgeable about the build process. Our approach enables the use of change-specific dependencies in multi-language software systems and shortens build times. We implemented our approach in a productive build environment to show the feasibility and practicability in a user study.**

***Index Terms*—software modeling, build automation, change propagation**

I. INTRODUCTION

Modern software engineering practices such as continuous integration use builds as fast feedback mechanisms to validate the quality of software [5]. Build systems usually face two requirements: i) Building should be fast. ii) The build process should yield reliable and reproducible results. Existing build tools often allow structuring a system into build components and defining a dependency graph on them. This enables developers to run partial builds (e.g., the modified build component and all its dependents). However, the structural dependencies defined by the dependency graph can differ significantly from actual dependencies with regard to the nature of a change.

A small change to a large software system may cause long build times, when considering only a subset of the dependent build components may be sufficient to produce sound build results. Making use of this knowledge and building only the affected build components is referred to hereafter as a build shortcut. Building only those components which are affected by a specific change can save a lot of time. However, in large development teams, there is often only a small number of developers who can identify the build components that are actually affected by a change. As the evolution of the product affects the dependency graph, the change-specific dependencies are also subject to change, but keeping all team members up to date is a time-consuming task.

DOI reference number: 10.18293/SEKE2018-030

Existing build tools, like grexmk [1] or vroom [3], speed up build times by executing build tasks in parallel, but come at the cost of the employed hardware. Approaches such as jmake for Java [13] allow language-specific change propagation analysis. However, they can only be applied to a specific set of software projects. Approaches like pluto [6] offer extensive customization of build tasks, but at high integration costs.

In this paper we propose an approach which enables developers less-experienced with build dependencies to selectively build those components that are affected by a specific change. To achieve this, our approach utilizes build shortcuts. The build experts of a development team annotate their knowledge on change-specific dependencies in the model of the software's build architecture. This model and the change scenarios are input for an existing change propagation approach for software architectures – Karlsruhe Architecture Maintainability Prediction (KAMP) [12]. The change propagation algorithm of KAMP identifies the build components to be built in order to implement the change. This way, we avoid rebuilding unaffected build components and consequently shorten the build times. Our model-based approach allows the formulation of build shortcuts for multi-language projects. Due to its simple interface, it can be used to extend an existing build system with little effort. The content of this paper has been developed in the context of the thesis *KAMP for Build Avoidance on Generation of Documentation* [9]. We evaluated our approach in a productive build environment by means of a user study. Our approach produced builds that were up to 27 times faster than building with the established build tool. The subjects of the user study assessed the approach as practicable.

The following section illustrates the PTV xServer, where our approach was applied. The foundations are described in Section III. Section IV gives an overview of the related work. We present our approach for build avoidance in Section V. In Section VII, we describe the results of our evaluation. This paper is concluded with a summary and an outlook on future work in Section VIII.

II. THE PTV XSERVER

We use the PTV xServer – a product of PTV Group – as a running example in this paper. The PTV xServer provides logistic and geographic solutions such as geocoding and trip-planning. PTV xServer is currently being developed by thirty developers from Germany and France.

The Application Programming Interface (API) of the server is described in so-called XServer Interface Description Language (XSIDL) files, which are used for the generation of C++ and Java source files. Furthermore, the comments in the XSIDL files are used in front-end components of the xServer to generate the product documentation for the customers. To build the xServer, an in-house build tool (the so-called b-Tool) is used, which is based on Apache Maven[1]. Apache Maven utilizes a dependency graph to identify the affected build components (called Maven projects) by a change. According to those dependencies, a change to one of the XSIDL files (e.g., a change the comments) requires a large part of the product to rebuild. Currently, this build process takes around four hours on average. However, rebuilding just the affected front-end components would only take a few minutes.

To avoid rebuilding unaffected build components, we use KAMP to analyze the scope of the change propagation in the PTV xServer. For this purpose, we modeled seven build shortcuts to consider various change scenarios. Furthermore, we extended the b-Tool by a new build command, called `shortcut`.

III. Foundations

Our approach is based on KAMP to identify the relevant build components affected by a specific change request. The KAMP approach aims to "analyse the change propagation caused by a change request in a software system based on the architecture model" [12]. For this purpose, KAMP uses a change propagation algorithm, which is validated through an empirical study [12]. Starting with the initial change request, the change propagation algorithm calculates the affected elements in the architecture model.

To model the architecture of a component-based software system, a Palladio Component Model (PCM) [10] can be used. In a PCM, the parts of a software system such as interfaces, components and their relations are modeled [10].

IV. State of the Art

The build tool grexmk [1] splits monolithic builds into multiple sand-boxed "mini-builds", taking advantage of loose coupling between system components. By executing these mini-builds incrementally or simultaneously, overall build times are shortened. The Vroom approach [3] analyzes the dependencies of test cases and carries out long-running tests in parallel. Although parallelization shows immediate results and is scalable by increasing hardware resources, it comes at the cost of this additional hardware. Instead of reducing time through parallelization, our approach saves time by not building unaffected components in the first place. Subsequently, the workload on build machines is lowered.

The build system pluto "supports the definition of reusable, parameterized, interconnected builders" [6]. As it abstracts from the programming language, pluto is applicable to a wide variety of projects. It can extract specific information from a file through domain-specific stampers. These file stampers can be used to implement change-specific rebuild behavior. But especially for large-scale projects, pluto would require a lot of implementation work, which also brings in additional risks. Our approach can coexist with a preexisting build environment by extending it, without introducing potentially breaking changes.

Smith describes the concept of smart dependencies [13]. This technique analyzes whether dependents need to be rebuilt by applying knowledge of the language-specific change propagation. The tool jmake [13, 8] applies smart dependencies for Java. If it detects modifications to publicly visible parts (e.g., method signatures), it rebuilds the dependent files. The addition of a new method or changes to comments, private methods, or the code within a method only requires a rebuild of the modified file itself. A build tool based on smart dependencies is language-specific, therefore it cannot be applied to multi-language projects. Our approach views dependencies between build artifacts on a more abstract level and is language agnostic. This comes with the disadvantage that shortcuts for specific change scenarios have to be defined manually and cannot be automatically derived.

V. Approach for Build Avoidance

Our approach allows the developers of a software system to build only the subset of those build components which are affected by a change. Figure 1 compares our build approach using build shortcuts (right) and the build process by Apache Maven (left). Apache Maven rebuilds the whole subgraph of the dependent build components, while our approach allows rebuilding only a specific subset of this subgraph.

The input of our approach is a PCM of the software system's build architecture, which contains the build shortcuts and the initial change requests. In the next step, KAMP is used to identify the build components which are potentially affected by the initial change requests. The result is a set of potentially affected build components. Deploying KAMP as a web service allows a loosely coupled build tool to access the build information. This way, only the model in the web service needs to be updated in order to change a dependency or add a new build shortcut. In the following subsections, we discuss our approach in more detail.

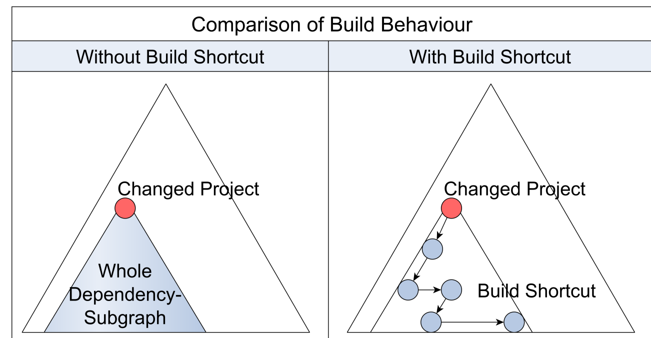

Fig. 1. Comparison of build behavior without and with build shortcut [9]

[1] http://maven.apache.org

A. Modeling of Build Shortcuts using Palladio Component Model

The dependency of a build component may be change-specific. Translated into a PCM, a component can provide several interfaces, which are required by different subsets of its dependent components. If a change to one of these interfaces is made, only the components that require this interface are potentially affected by the change (i.e., not all components that require interfaces of a specific component) [12].

Figure 2 shows a simplified example from the PTV xServer. The change scenarios of the component `model` are represented by the two interfaces `model` and `documentation`. The `model` and `documentation` interfaces are required by `frontend` component. The `model` interface is required by `services` and `runtime` components. Let us assume that `documentation` interface is changed. In this case, only the `frontend` component is affected by the change, and will not affect `services` and `runtime` components.

Our approach uses a PCM to model the build architecture, which can differ from the software architecture. Originally, PCM was designed to model component-based software architectures. However, it can also be used to model the build architecture, which describes relations between the build components, thus providing a more technical view on the software system.

A build component can contain several software components, as illustrated in Figure 2. Although the build component may provide more than one functionality, it cannot always be divided into subcomponents in accordance to the change scenarios. In the example of the `model` component in the PTV xServer, the separation of the documentation from the API model may cause the developers to neglect keeping the documentation up to date.

B. Utilization of KAMP's Change Propagation Analysis

As described above, the build architecture can differ from the software architecture. The Apache Maven projects can be mapped to components in PCM, while different types of change scenarios for each Maven project represent different interfaces. Based on an initial change request, KAMP uses its change propagation algorithm to identify the affected build components [12]. KAMP's change propagation algorithm consists of a set of change propagation rules [12]. An example of a rule is the change propagation from a modified interface to all components that provide or require this interface [12]. Applied to the build architecture, this method allows identifying the affected build components. By utilizing the domain knowledge annotated in the PCM, KAMP can reduce the set of build components that are rebuilt.

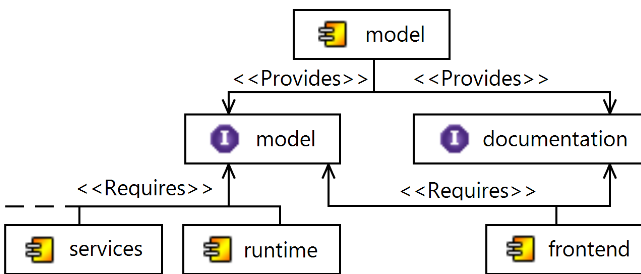

Fig. 2. Modeling of a build shortcut with PCM [9]

VI. Prototypical Implementation

Figure 3 illustrates the deployment of our approach at PTV Group. The components of our approach run on a server (i.e., the server side) in the PTV intranet and on the individual developer workstations (i.e., the client side). On the server side, the `KAMP-WS Server` is deployed together with the `PCM` of the software under development (i.e. the PTV xServer PCM) and `KAMP`. On the client side, the build tool (i.e., PTV b-Tool) is deployed. The PTV b-Tool is extended by the command `shortcut`, which uses the web service.

When a build with the `shortcut` command is triggered, the available change scenarios for the modified Maven project are requested from the KAMP web service. This list is presented to the user, who selects the appropriate scenario over the command line interface. Then, a second request is sent to the KAMP web service querying the dependent projects for the selected change scenario. After `shortcut` receives the response, it triggers an Apache Maven build of specifically the dependent projects.

VII. Evaluation

To evaluate our approach, we follow the Goal Question Metric (GQM) [2] plan. We define two evaluation goals:

- **Feasibility** aims to "validate the prediction accuracy [...] by comparing the prediction results to reference values" [7]. It is assumed that inputs are correct.
- **Practicability** aims to "validate the practicability of a method, when it is applied by target users, instead of method developers" [7].

In the following, we define research questions (RQ) for each evaluation goal. Furthermore, we propose the metrics to evaluate the corresponding evaluation goals and research questions.

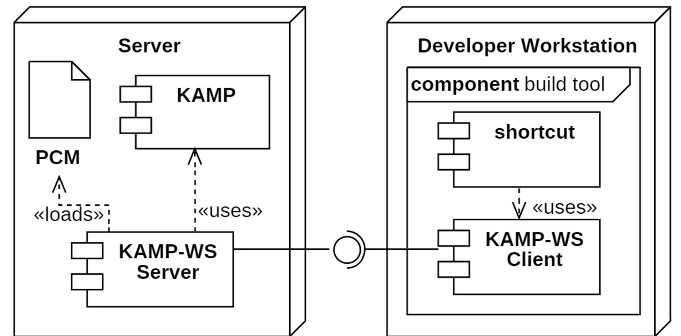

Fig. 3. Deployment diagram of the system [9]

A. Feasibility

To evaluate the feasibility of our approach, we compare the actual outputs of our approach to expected results. The build experts provided reference lists for a total of seven build shortcuts, each containing the build components that have to be actually built to validate a change scenario. We annotated the respective build shortcuts in the xServer PCM used by the web service. Thus, we need to check the equality of the two sets for each change scenario: the set of Maven projects proposed by our approach (i.e., $\mathrm{P}_{\mathrm{shortcut}}$), and the set of Maven projects in the corresponding reference list (i.e., $\mathrm{P}_{\mathrm{ref}}$). In other words, the following formula should be true for each of the seven change scenarios:

$$\mathrm{P}_{\mathrm{shortcut}} = \mathrm{P}_{\mathrm{ref}} \Leftrightarrow \mathrm{P}_{\mathrm{ref}} \subseteq \mathrm{P}_{\mathrm{shortcut}} \wedge \mathrm{P}_{\mathrm{shortcut}} \subseteq \mathrm{P}_{\mathrm{ref}}$$

This leads to the following research questions and hypotheses:

RQ 1: Does the output of our approach include all build components that have to be built for the respective change scenario?

Hypothesis **H1** is that the output of our approach includes all build components that have to be built according to the reference lists (i.e., $\mathrm{P}_{\mathrm{ref}} \subseteq \mathrm{P}_{\mathrm{shortcut}}$). In other words, we assume that the build components that are necessary to validate the change scenario are included.

RQ 2: Does the output of our approach not include build components that are unaffected by the respective change scenario?

Hypothesis **H2** is that the output of our approach does not include build components that are not on the reference list of the change scenario (i.e., $\mathrm{P}_{\mathrm{shortcut}} \subseteq \mathrm{P}_{\mathrm{ref}}$).

We designed a set of system tests to gather metrics to verify hypotheses **H1** and **H2**. Two types of tests exist for each change scenario. The first test checks whether our approach builds all build components specified in the reference lists. The second test checks whether the build does not include an unaffected build component. If both tests are successful, the list of build components that are built with our approach for a given change scenario is the same as the corresponding reference list.

As described above, we implemented our approach as a new build command `shortcut` and added it to PTV's b-Tool. We want to verify that building with the `shortcut` command takes less time than building with `down`, which is an established b-Tool command of the xServer development team and builds a build component and its dependents.

RQ 3: Is the duration of a build process triggered with the `shortcut` command shorter than the duration of a comparable build process triggered with the `down` command?

Hypothesis **H3** is that the duration of a build triggered with `shortcut` is shorter than the duration of a comparable build triggered with `down`.

To answer this research question we define the execution time metric for both `shortcut` and `down` commands and compare the build execution times measured by developers on their workstations, as described in the following.

B. Practicability

Our evaluation of practicability is based on the Technology Acceptance Model (TAM) [4]. According to TAM, the intentions of a person to use a technology determines the actual use [4, 9]. For each of TAM's variables, a research question and a hypothesis was formulated, taken from [9]:

RQ 4: Do the subjects find the activities provided by our approach (i.e., `shortcut` command) important?

Hypothesis **H4** is that the subjects find the activity provided by our approach (i.e., building only the affected Maven projects in a change scenario) important.

RQ 5: Does our approach ease performing the activities?

Hypothesis **H5** is that our approach eases performing the activities (i.e., Perceived Usefulness [4]).

RQ 6: Is our approach easy to use in practice?

Hypothesis **H6** is that our approach is easy to use in practice (i.e, Perceived Ease of Use [4]).

RQ 7: Do the subjects have a specific intention to use our approach?

Hypothesis **H7** is that the subjects have a concrete intention to use our approach (i.e., Behavioral Intention to Use [4]).

RQ 8: Do the subjects expect concrete consequences by using our approach?

Hypothesis **H8** is that the subjects expect concrete consequences by using our approach (i.e., Attitude Toward Using [4]).

We conducted a user study to evaluate the previously described research questions and validate the respective hypotheses. The subjects of the study were the potential users of our approach at PTV. They evaluated the usage and performance of the `shortcut` command in comparison to the `down` command.

C. User Study Design

The user study consisted of a task sheet and a questionnaire. In the task sheet, the subjects were asked to run these three different builds on the `model` Maven project (cf. Section II and Figure 2):

- Trigger the build process using `down` command (hereafter referred to as $down$).
- Trigger the build process using `shortcut` with the model change scenario (hereafter referred to as $shortcut_{model}$).
- Trigger the build process using `shortcut` with the documentation change scenario (hereafter referred to as $shortcut_{doc}$).

We chose these build scenarios because most members of the development team are familiar with this build component and have worked on it before. Thus, the change scenarios were well understood by the subjects.

The subjects were asked to provide the resulting build times in the questionnaire. Furthermore, the subjects provided information about their usual build behavior and how they evaluate our approach.

The questionnaires contained a set of free text questions. Also, the subjects had to rate the following statements, taken from [9], on a 6-point Likert scale, where 1 means ”I strongly disagree” and 6 means ”I strongly agree”:

S1 I frequently validate changes through local builds.
S2 Building (parts of) the xServer is important for my work.
S3 I could work more efficiently if local builds were faster.
S4 Local builds take too long.
S5 I try to only build the projects that were affected by my change.
S6 `shortcut` command will be useful to me.
S7 `shortcut` command is easy to use.
S8 I am motivated to use `shortcut` command for my local builds in the future.

We aim to assess the importance of the build process to the subjects with statements **S1** and **S2**. Statements **S3 – S5** are included to verify that the subjects are dissatisfied with the current build times. The remaining statements **S6 – S8** aim to evaluate **RQ 5 – RQ 7**.

The xServer team consists of 30 Java and C++ developers, of which 18 participated in our study. Note that the C++ parts of the xServer have longer build times than the Java parts. An evaluation of the answers is presented in the following.

D. Results

System Test Results: The system tests for each of the seven build shortcuts were successful. In other words, the statements $\mathrm{P}_{\mathrm{ref}} \subseteq \mathrm{P}_{\mathrm{shortcut}}$ and $\mathrm{P}_{\mathrm{shortcut}} \subseteq \mathrm{P}_{\mathrm{ref}}$ hold true for each shortcut. That confirms **H1** and **H2**. Thus, we can conclude $\mathrm{P}_{\mathrm{shortcut}} = \mathrm{P}_{\mathrm{ref}}$.

User Study Builds: The subjects were asked to provide the build times for the builds they had to trigger. Table I gives an overview of the build times. The $down$ build scenario took longest (i.e., from about 1.5 hours to almost 9 hours). The builds of the $shortcut_{model}$ scenario took between 30 minutes and almost 2.5 hours, while the builds of $shortcut_{doc}$ took only 3 to 12.5 minutes. On average, the builds of $shortcut_{model}$ were 3.45 times faster than the builds of $down$, while the builds of the $shortcut_{doc}$ scenario were 27.07 times faster than $down$.

The significant difference in build times for a single build scenario can be accounted to different hardware of the developer workstations and network latency, as some subjects used a VPN connection from France. The build execution times of

TABLE I
EVALUATED BUILD TIMES FOR THE SCENARIOS IN THE USER STUDY [9]

	$down$	$shortcut_{model}$	$shortcut_{doc}$
Max	8h 55min	2h 27min	12min 30s
Avg	3h 42min	1h 1min 30s	9min 4s
Min	1h 32min	30min 6s	2min 59s

TABLE II
DETAILED RATINGS FOR STATEMENTS S1-S8 IN THE USER STUDY [9]

	1: Strongly disagree	2: Disagree	3: Rather disagree	4: Rather agree	5: Agree	6: Strongly agree
S1:	0	0	0	0	2	17
S2:	0	0	0	0	3	16
S3:	0	0	0	2	4	13
S4:	0	0	1	2	7	9
S5:	0	0	0	1	5	13
S6:	0	0	2	1	3	13
S7:	0	0	1	1	4	13
S8:	0	1	1	1	3	13

the $shortcut_{model}$ and $shortcut_{doc}$ scenarios are lower than the build time of $down$. This confirms **H3**.

Activity Importance: Statements **S1** and **S2** aim to assess the importance of builds to the subjects, as illustrated in Table II. All subjects rated statements **S1** and **S2** with 5 (i.e., agree) or 6 (i.e., strongly agree), which confirms **H4**.

Satisfaction with Builds: Table II shows how the subjects rated statements **S3–S5**. Those subjects who rated statement **S4** with 4 or lower contribute mainly to Java parts of the xServer, which have lower build times in general. We conclude that developers perceive long builds as a problem and they try to shorten them.

Usefulness: Statement **S6** aims to assess the perceived usefulness of our approach. As seen in Table II, most subjects rate our approach as useful. This confirms **H5**. Only two Java developers specified that they rather disagree with the statement. One of them noted that they work on a partial checkout of the xServer repository. They could not use our approach, because the utility functions used by the implementation of `shortcut` work under the assumption of a full checkout. We did not consider this requirement (i.e., working on partial checkouts) during development of our approach, but it could be supported in the future.

Ease of Use: Most subjects found our approach easy to use (i.e., statement **S7**), as seen in Table II. This confirms **H6**. Eight subjects additionally mentioned in the free text fields that they found our approach easy and intuitive to use. One subject rated statement **S7** with 3 (i.e., rather disagree), however, gave no further comment in the free text fields.

Intention To Use: Most subjects rated Statement **S8** (i.e., using our approach `shortcut` for the local builds in the future) with 4 or higher. That confirms **H7**. One subject, who works on the Java parts of the xServer, rather disagreed with

this statement. Also, the subject who stated that they work on partial checkouts disagreed with the statement.

Attitude Toward Using: The questionnaire contained two free-text questions regarding the advantages and disadvantages of our approach. We can derive from the given answers which consequences the subjects expect by using our approach. The following advantages were expected by the subjects, taken from [9] (The number in parentheses states how often it was mentioned by the subjects):

- Faster builds (10)
- Fewer unnecessary builds (6)
- Increased work efficiency (3)
- Fewer failed builds (2)
- Less knowledge about dependencies required (2)
- Conveniently trigger subtarget builds (2)
- Faster validation of code / Earlier detection of defects (2)
- Sharing of useful build shortcuts
- No more manual triggering of different artifacts

The following disadvantages were expected by the subjects, taken from [9]:

- Maintenance cost of build shortcuts (7)
- Shortcut names may become confusing (2)
- Far-reaching consequences if a shortcut is incorrect (2)
- No batch-compatibility
- Less pressure to solve dependency problems
- Less pressure to optimize build times
- Developer's knowledge about dependencies decreases

Each subject stated at least one expected consequence. This confirms **H8** (i.e., subjects expect concrete consequences by using `shortcut`). The number of advantages mentioned is higher than the disadvantages. In other words, subjects expected predominantly positive consequences.

By the results of our user study, we see both the feasibility and the practicability of our approach confirmed.

VIII. Conclusion and Future Work

In this paper, we presented an approach to shorten the build times, based on build shortcuts. It makes the advantages of build shortcuts also accessible to those members of the development team who are less knowledgeable about the build process.

The change-specific dependencies are modeled in a PCM. The model of the build architecture and the change requests are used as input for the KAMP approach. The KAMP approach uses a change propagation algorithm to identify the affected build components for a given change scenario.

We evaluated our approach using a set of system tests and a user study. The system tests, which checked whether the builds produced by our approach are conform to the reference lists created by build experts, were all successful.

In the user study, 18 developers were asked to compare the results of three different build scenarios. The execution time of the build process for the model scenario was on average about 3.5 times faster than the reference build with the `down` command, while the documentation build was 27 times faster.

In the questionnaires of the study, the subjects indicated that they found the new command useful and have the intention to use it in the future. We see both feasibility and practicability of our approach confirmed through the results of the evaluation.

The KAMP web service continues to be used and maintained in the PTV xServer project. Much of the feedback received in the user study was already incorporated (e.g., the extension of `shortcuts` API to support batch-compatibility). Furthermore, additional shortcuts were added to the model.

As future work, we plan to extend the KAMP web service to assist development teams with other tasks such as analyzing the change propagation of backlog items to coordinate the work of multiple scrum teams in a scrum of scrums.

Acknowledgment

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593 (RE1674/12-1).

References

[1] Glenn Ammons. "Grexmk: speeding up scripted builds". In: *International workshop on Dynamic systems analysis*. ACM. 2006, pp. 81–87.

[2] Victor Basili et al. "The Goal Question Metric Approach". In: *Encyclopedia of Software Engineering* 2.1994 (1994), pp. 528–532.

[3] Jonathan Bell et al. "Vroom: Faster Build Processes for Java". In: *IEEE Software* 32.2 (2015), pp. 97–104.

[4] Fred Davis et al. "User acceptance of computer technology: a comparison of two theoretical models". In: *Management science* 35.8 (1989), pp. 982–1003.

[5] Paul Duvall, Stephen Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[6] Sebastian Erdweg et al. "A sound and optimal incremental build system with dynamic dependencies". In: *SIGPLAN Notices*. Vol. 50. ACM. 2015, pp. 89–106.

[7] Robert Heinrich. *Aligning Business Processes and Information Systems: New Approaches to Continuous Quality Engineering*. Springer, 2014.

[8] JMake. *JMake*. https://github.com/pantsbuild/jmake. [Online; accessed April 2018]. 2014.

[9] Milena Neumann. "KAMP for Build Avoidance on Generation of Documentation". Bachelor's thesis. Karlsruhe Institute of Technology, 2017.

[10] Ralf Reussner et al. *Modeling and simulating software architectures: the Palladio approach*. MIT Press, 2016.

[11] Kiana Rostami et al. "Architecture-based Change Impact Analysis in Information Systems and Business Processes". In: *ICSA2017*. IEEE, 2017, pp. 179–188.

[12] Kiana Rostami et al. "Architecture-based Assessment and Planning of Change Requests". In: *11th International ACM SIGSOFT Conference on Quality of Software Architectures*. ACM, 2015, pp. 21–30.

[13] Peter Smith. *Software Build Systems: Principles and Experience*. First. Addison-Wesley Professional, 2011.