Communicating Continuous Integration Servers for Increasing Effectiveness of Automated Testing

Stefan Dösinger, Richard Mordinyi, Stefan Biffl
Christian Doppler Laboratory "Software Engineering Integration for Flexible Automation Systems"
Vienna University of Technology
Vienna, Austria
{firstname.lastname}@qse.ifs.tuwien.ac.at

ABSTRACT

Automated testing and continuous integration are established concepts in today's software engineering landscape, but they work in a kind of isolated environment as they do not fully take into consideration the complexity of dependencies between code artifacts in different projects. In this paper, we demonstrate the Continuous Change Impact Analysis Process (CCIP) that breaks up the isolation by actively taking into account project dependencies. The implemented CCIP approach extends the traditional continuous integration (CI) process by enforcing communication between CI servers whenever new artifact updates are available. We show that the exchange of CI process results contribute to improving effectiveness of automated testing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms

Experimentation

Keywords

Software testing; test coverage; software libraries; software project dependency; dependency management

1. INTRODUCTION

Typical software engineering projects take advantage of the continuous integration (CI) process as an approach to automate certain tasks for the purpose of improving software quality, like the execution of tests. However, testing mainly focuses on the code artifacts which were implemented according to project requirements without taking into account imported code artifacts from other projects which may be developed outside the organization or engineering domain. Therefore, projects' code artifacts are tested in a kind of isolated environment as they do not fully take into account the complex dependencies between different projects. While project dependencies (i.e. projects a project depends on) are implicitly handled by build management mechanisms, project dependents (i.e. projects which depend on a project) on the other hand are hardly considered if at all.

However, a dependency implies that changes in a project's code artifact affects the implementation of dependent projects as well, resulting in potentially failing tests there. Upgrading dependencies, finding problems and reporting them to their authors is done manually and is a time-consuming task. Likewise, getting feedback from dependent projects is slow. Often problems are not detected until after a change has been completed and integrated into the live system, which leads to problems for developers and users of libraries alike [4].

In this paper, we will demonstrate the Continuous Change Impact Analysis Process (CCIP) as an approach that fully takes into account the complexity of dependencies between code artifacts of various software engineering projects to improve the effectiveness of automated tests. CCIP automatically collects feedback from dependent projects due to local code artifacts changes for further quality improvements before those updates become public. The approach relies on Continuous Integration servers exchanging information (e.g., about available updates or failed tests on potential releases) which allows the software engineers to analyze the impact of artifact changes on depending projects beforehand, and helps detect software development problems like regressions early and bring them to the developers' attention. This will automate routine communication between interdependent projects, increase software development quality, and decrease manual testing effort.

2. RELATED WORK

In this section we summarize related work on the continuous integration approach, impact analysis definitions and approaches, and software dependencies.

2.1 Continuous Integration Process

The strength of the Continuous Integration (CI) workflow [5] is the automation of sequential build process steps for the purpose of improving software quality and minimizing the time to delivery. Whenever a software engineer commits changes, the CI server checks out the code, builds the artifact, runs tests, and deploys the produced artifacts if configured (e.g., to provide nightly builds on an FTP server).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASE'12, September 3–7, 2012, Essen, Germany
Copyright 2012 ACM 978-1-4503-1204-2/12/09 ...$15.00

Regardless of the outcome of the workflow, the CI server may send notifications with the final result to software engineers. In case a step in the process fails, further process steps are skipped and an error notification is sent. There are a number of requirements to fulfill for running an efficient CI: the build process has to be fully automated, a revision control system is required, and an automated test-suite is necessary. Software engineers should commit their changes regularly and early. The build process and tests have to complete quickly enough to allow the CI server to keep up with the changes and return feedback to the software engineers while their memory of the changes is still fresh [6]. Nevertheless, testing in the CI process focuses only on "local" projects and does not take into account project dependents and their test scenarios to increase test coverage. Popular CI implementations like Jenkins[1], Hudson[2] and Buildbot[3] have rudimentary support for dependency tracking, but this only works between projects that are built on the same server and without the ability to communicate with other CI servers.

2.2 Impact Analysis

Impact analysis is defined either as "The activity of identifying what to modify to accomplish a change", or "the activity of identifying the potential consequences of a change" [1]. The former definition has been investigated by Canfora and Cerulo who developed a concept [2] and tool implementation [3] to predict which code files have to be changed to implement a change request. The tool evaluates resolved change requests and the changes that resolved them to isolate keywords which are used on new change requests to predict the code areas that need changes to resolve the new change requests. They demonstrated the approach using the change request and code history of a few selected open source projects. The latter usually comes down to identifying which parts of a program might be affected by a code change. The aim is to help the software engineer identify potential unintended side effects and to identify which test cases need to be rerun [10]. The shortcoming of these techniques is that they are only tools for a manual analysis of the impact of a change.

2.3 Software Dependencies

Reuse of software components, libraries etc. is a common software engineering practice. As a consequence, dependencies have to be provided in the correct version to the system, since dealing with dependency upgrades can be a major cost factor in software evolution [13]. While closed source operating systems are developed and controlled by companies, open source systems consist of many independently developed components [9]. Assembling the individual components results in complex dependency relationships, which have been analyzed in scientific studies [7,8]. A formal model for software dependencies has been proposed by Podgurski and Clarke [11]. Various tools have been developed to aid dependency tracking (e.g., [12]). The concept of dependency relationships provides a scientific base for our approach.

[1]http://www.jenkins-ci.org
[2]http://www.hudson-ci.org
[3]http://trac.buildbot.net

3. SOLUTION APPROACH FOR DEMONSTRATION

This section describes the concept of the CCIP and describes our tool implementation.

3.1 Conceptual Overview

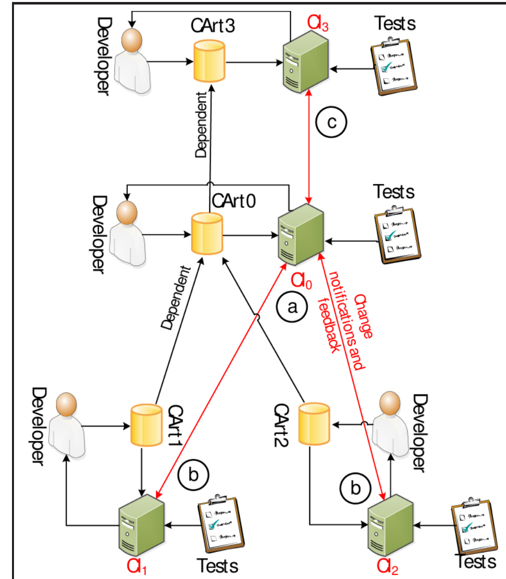

Figure 1: Flow of Communication in CCIP

Figure 1 presents the conceptual overview of the CCIP approach, which is based on the generic CI concept. The core difference to the traditional CI approach is that CI servers enter a two-way communication. The CI servers of the dependent projects register themselves with the CI servers of their dependencies. In this example setup, code artifact 0(CArt0) depends on CArt1 and CArt2. CArt3 in turn depends on CArt0. 1-a: CI Servers of project dependencies report new code releases after they have passed their own tests. Those new versions could be manually tagged numbered releases, they could be automated nightly builds or snapshots updated for each commit to the revision control system. 1-b: When the CI servers of the dependent projects receive a notification, they run their tests with the new version of the dependency in a cloned environment and report their test results back to the CI servers of the dependency. 1-c: Because the process is cascading, CArt0 can report its new build to its dependent projects and forward test results to the originator of the change.

The intended effect is that the tests of dependent projects are automatically used to increase the test coverage of the dependency and improve communication between the projects' development teams.

Figure 2 outlines the CCIP workflow. At the core of the CCIP is a publish-subscribe message system to send update notifications and feedbacks. It introduces the following changes to the standard CI workflow:

1. Merge: In addition to changes to the project's own source code, the CCIP workflow can be triggered by dependency update notifications. The dependencies have to be integrated into the build environment. For

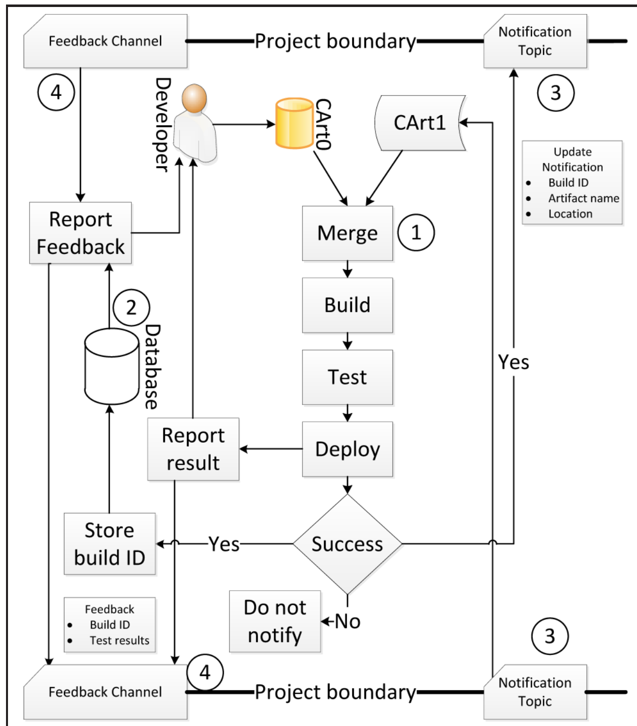

Figure 2: The CCIP workflow

more details please refer to section 3.3. The CI server has to keep track of the origin of the change to be able to send the feedback to the correct destination.

2. Build Database: In order to handle feedback correctly, the CI server has to store information about old builds, most importantly which change triggered the build. Most CI servers already do this for other reasons, like allowing the user to look up the build history.

3. Extended notification: On successful builds a notification message is sent via the update notification topic. Other CI servers can subscribe to this topic to receive them. If the build fails, no public notification is sent, but the regular project-internal build report is still generated. See section 3.2 for details about these messages.

4. Feedback handling: If a build was triggered by an update notification, the builds result(positive or negative) is sent to the feedback queue of the CI server that sent the update notification. For details see section 3.2.

To initiate the process, the CI server of the dependent code artifact has to make contact to the CI servers of its dependencies. Ideally, the necessary information can be extracted from the existing build system configuration if the build system supports dependency management and automatic dependency downloading. Otherwise the administrator of the CI server has to provide the necessary information manually.

3.2 Communication Protocol

As described before, the CCIP relies on communicating CI servers. Therefore, it is necessary to clarify the type and amount of information exchanged between the various servers. An update notification needs at least the following information:

- A unique build ID. This is necessary to relate feedback to builds.
- The artifact name, unless a unique topic is used for each artifact.
- Information where to obtain the artifact. This information depends on the build system and cannot be generalized. Considerations regarding artifact transfer and merging are described in 3.3.

To improve the usability, a few optional fields can be added:

- Whether new build or test failures are expected compared to the previous version, e.g. if the API was changed.
- What kind of build it is, e.g. an automated snapshot build, a release candidate, a minor release or a major release. This allows the recipients to limit test runs to builds they are interested in.
- Human readable version and release notes.

A feedback message from project dependents contains:

- The build ID to which this feedback is a response to.
- The build result: Build failure, test failure, success, failure in a dependent project.
- Debugging information like logging output in the case of a failed build.
- Contact information of the feedback sender.

In publish-subscribe communication pattern the publisher does not know the subscribers. This basically works for the CCIP too, but for better usability and improved evaluation metrics additional information in accordance with project's policies can be sent when the subscription is initiated:

- Identity of the subscriber and contact information.
- Which notifications the subscriber will act upon.
- Whether the subscriber intends to provide feedback.
- Expected response time.

Obviously it is up to the receiver of an update notification which information he sends or if he sends feedback at all. Privacy may be a concern here as test names or test output may reveal confidential information.

3.3 Artifact Transfer and Merging

How artifacts are transfered and integrated depends on the kind of the artifact and the build system the project uses. Some build systems like Apache Maven[4] download dependencies automatically. In this case the only necessary information is the new version. To merge the new version into the dependent project, the merge step merely has to adjust the version information in the Maven configuration files. On the other hand, a dependency like an operating system kernel or device driver may make a reboot of the test machine necessary.

As a consequence, it is not possible to define the layout of the download information and the merge step in a way that will work for every software artifact. To keep the setup effort low for average projects it may be conceivable to provide templates for popular build systems and handle the remaining cases with user-written scripts.

4. DISCUSSION

As mentioned before, the CCIP is an extension of the CI process. It provides an easy way to extend test coverage for more effective testing and improved communication between dependencies and their dependent projects. The amount of improvement depends on the involved projects, especially the quality of the testsuite in the dependent projects and the heterogenity of environments where the dependency is used. The CCIP is not intended to replace manual impact analysis, but to augment them. The CCIP can provide developers with automated testing results for faster defect detection and localisation. On the other hand, CCIP implementations can use existing impact analysis techniques and network metrics to prevent unnecessary test runs.

5. CONCLUSION AND FUTURE WORK

The continuous integration process implemented in several continuous integration servers is a well-accepted approach in today's software engineering landscapes. However, since the process only focuses on local repositories, automated testing is as powerful as the tests implemented by the project's software engineers. In this paper, we demonstrated the Continuous Change Impact Analysis Process (CCIP) approach that takes into account project dependents and makes use of test scenarios implemented there. The approach introduces communicating continuous integration servers which exchange information about tests results of potential releases and thus increase the effectiveness of automated testing. Further work considers extending the implementation to other build and source control tools and investigating the behaviour under heavy load and handling of poor quality feedback. Furthermore, we will investigate the improvements the CCIP concept provides to the software development process. Our expectation is that the CCIP can improve the effectiveness of testing, especially in projects that are used in heterogeneous environments.

6. ACKNOWLEDGMENT

This work has been supported by the Christian Doppler Forschnungsgesellschaft and the BMWFJ, Austria.

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

7. REFERENCES

[1] R. Arnold and S. Bohner. Impact analysis-towards a framework for comparison. In *Software Maintenance ,1993. CSM-93, Proc., Conf. on*, pages 292 –301, sep 1993.

[2] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Software Metrics, 2005. 11th IEEE Int. Symposium*, pages 9 pp. –29, sept. 2005.

[3] G. Canfora and L. Cerulo. Jimpa: An eclipse plug-in for impact analysis. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proc. of the 10th European Conf. on*, pages 2 pp. –342, march 2006.

[4] C. R. de Souza, S. Quirk, E. Trainer, and D. F. Redmiles. Supporting collaborative software development through the visualization of socio-technical dependencies. In *Proc. of the 2007 int. ACM conf. on Supporting group work*, GROUP '07, pages 147–156, New York, NY, USA, 2007. ACM.

[5] P. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[6] M. Fowler. Continuous integration. `http://martinfowler.com/articles/continuousIntegration.html`, 2006.

[7] D. German. Using software distributions to understand the relationship among free and open source software projects. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth Int. Workshop on*, page 24, may 2007.

[8] N. LaBelle and E. Wallingford. Inter-package dependency networks in open-source software. *CoRR*, cs.SE/0411096, 2004.

[9] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM Int. Conf. on*, pages 199 –208, sept. 2006.

[10] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proc. of the 26th Int. Conf. on Software Engineering*, ICSE '04, pages 491–500, Washington, DC, USA, 2004. IEEE Computer Society.

[11] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *Software Engineering, IEEE Transactions on*, 16(9):965 –979, sep 1990.

[12] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. of the 20th annual ACM SIGPLAN conf. on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 167–176, New York, NY, USA, 2005. ACM.

[13] H. Sneed. A cost model for software maintenance evolution. In *Software Maintenance, 2004. Proc. 20th IEEE Int. Conf. on*, pages 264 – 273, sept. 2004.