2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

CodeAware: Sensor-Based Fine-Grained Monitoring and Management of Software Artifacts

Rui Abreu
Palo Alto Research Center
Palo Alto, CA 94304
Email: rui@computer.org

Hakan Erdogmus
Carnegie Mellon University - Silicon Valley
Moffett Field, CA 94035
Email: hakan.erdogmus@sv.cmu.edu

Alexandre Perez
Palo Alto Research Center
Palo Alto, CA 94304
Email: alexandre@computer.org

***Abstract*—Current continuous integration (CI) tools, although extensible, can be limiting in terms of flexibility. In particular, artifact analysis capabilities available through plugin mechanisms are both coarse-grained and centralized. To address this limitation, this paper introduces a new paradigm, CodeAware, for distributed and fine-grained artifact analysis. CodeAware is an ecosystem inspired by sensor networks, consisting of monitors and actuators, aimed at improving code quality and team productivity. CodeAware's vision entails (a) the ability to probe software artifacts of any granularity and localization, from variables to classes or files to entire systems; (b) the ability to perform both static and dynamic analyses on these artifacts; and (c) the ability to describe targeted remediation actions, for example to notify interested developers, through automated actuators. We provide motivational examples for the use of CodeAware that leverage current CI solutions, sketch the architecture of its underlying ecosystem, and outline research challenges.**

I. Introduction

The roots of continuous integration (CI) can be traced back to the Extreme Programming (XP) methodology [4]. Among the objectives of CI is to alleviate the problems of "integration hell", the street term that refers to different engineers simultaneously working on the same *codebase* having to eventually merge their conflicting changes to make the application work. A common way to deal with such problems is to use source code management (SCM) tools (e.g., SVN, Git, or Mercurial) and to frequently commit changes. At each new code commit, a remote server (also known as *build server* or *CI server*) pulls the code from the version control system and builds the application automatically to determine if there are any merge conflicts. Furthermore, at each new build, specified regression test suites can be run to see if any new features or bug fixes break existing functionality. Various forms of static (e.g., FindBugs [2]) and dynamic analyses (e.g., code coverage [1]) can also be included in the build pipeline through third-party plugins. The development team can be automatically notified of successful builds or failures in the build pipeline. Fine-tuning is possible to filter the results or to confine analyses to a portion of the codebase. However this is achieved through centralized configuration of the build server and its plugins.

CI is widely adopted in industry, and several different systems are available to practitioners. The most popular ones include the open source projects Jenkins,[1] CruiseControl,[2] Apache Continuum,[3] Oracle's Hudson,[4] and Bamboo from Atlassian.[5] The functionalities of these CI systems can typically be extended with plugins. For example, at the time of writing this paper, Jenkins had more than 600 plugins, including plugins to perform static and dynamic analyses, such as style checking and measuring test-case coverage.

Localized approaches rely on plugins attached to instances of integrated development environments (IDEs) running on the engineers' local hardware. The plugins, or loggers, collect data from an engineer working on a software project and either directly visualize the information in the engineer's environment or send the collected data to a central repository for further, offline analysis. Engineers' code navigation history, test runs, files changed, and file metrics can be recorded [9], [8]. Project-level metrics can also be tracked by aggregating data from multiple engineers. Hackystat [6] and AnalyzeD [7] are examples of such systems. Whether the data are stored and processed locally or centrally, relevant information about what others engineers are doing and how their actions may impact a specific engineer is still missing.

Therefore, while these systems and the plugins available for them are very useful, the underlying approaches employed to monitor and manage code artifacts are monolithic and too coarse-grained to be scalable (see Figure 1). In addition engineers cannot register interest only in code artifacts that affect them. When the codebase is large, this centralized, untargeted approach may become too inflexible and lose its usefulness. We wish to flip this approach on its head using a sensor network metaphor, allowing quality monitoring and management of software artifacts to be distributed, granular, and individually definable, controllable, and actionable.

Thus CodeAware (see Figure 2) is a new research vision based on a flexible, scalable, and extensible ecosystem of soft agents that support software engineers in their endeavor to improve code quality and team productivity. The ecosystem is composed of probes, coordinators, actors, and dashboards (and variants thereof) that may be attached to and deployed with

[1]http://jenkins-ci.org, accessed Jan, 2015.
[2]http://cruisecontrol.sourceforge.net, accessed Jan, 2015.
[3]http://continuum.apache.org, accessed Jan, 2015.
[4]http://hudson-ci.org, accessed Jan, 2015.
[5]http://atlassian.com/software/bamboo, accessed Jan, 2015.

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.192

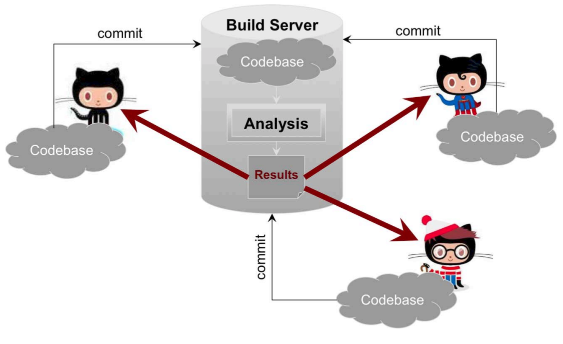

Fig. 1. The classical approach: monolithic, centralized, inflexible

individual code fragments such as statements, blocks, methods, tests, files, and directories, as well as logical artifacts such as variables, classes, data types, packages, and aspects.

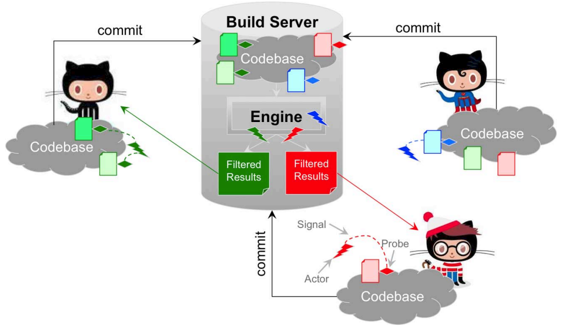

Fig. 2. The CodeAware approach: fine-grained, distributed, flexible/targeted

These soft sensors, controllers, actuators, and dashboards are user definable and may have different purposes. Some may be detectors that perform static analyses to identify common risky coding patterns or collect complexity metrics about the artifacts to which they are attached. Others may monitor changes (e.g., about execution or commit time of a given method). Meta-probes may aggregate input from multiple probes to reveal higher level information. Coordinators may subscribe to input from a variety of probes to devise actionable strategies when corrective measures are warranted. Certain coordinators may be responsible for the meta-monitoring of the ecosystem so that the constituents live in harmony and without conflict and can be searched, enabled, disabled, spread, constrained, deleted, and maintained easily. These strategies are deployed through actors that may perform a myriad of missions such as automatically fire alerts, generate reports, allocate or free resources, update plans, make changes to underlying software artifacts, log issues in issue tracking systems, or flag software artifacts for review. Dashboards allow subscribers to visualize the collected information in a digestible form and give them access to actors. Instead of installing a logger on a local machine and define filters to restrict the information collection, engineers define and deploy these agents incrementally for selected software artifacts.

The ecosystem grows organically with the code base and according to evolving needs as new types of probes and actors are defined. The resulting approach is consequently more evolutionary and robust than traditional big-bang approaches to software quality management.

Our hypothesis is that the CodeAware ecosystem will improve both productivity and software quality by bringing relevant changes, not only external ones caused by updates in dependencies [5] but also internal changes within the codebase, to the attention of the software engineer before the fact in a manageable and targeted way, thus emphasizing efficient and proactive prevention over fault localization and fixing. Our goal is to foster discussion around the metaphor underlying CodeAware and encourage future endeavors that take advantage of it.

II. Example Scenario

In this section we present an example scenario to motivate using CodeAware over current CI strategies. The scenario is explained through a narrative that captures CodeAware's vision. We take a declarative and linguistic approach to the narrative, but it is easy to imagine scenarios that can be executed through modern UI mechanisms, such as selecting, dragging, dropping, input widgets, and workflow wizards.

Let us assume that Sue, a game developer, is responsible for ensuring the game her team is developing runs at a certain speed. She knows that, in order to provide an enjoyable experience to players, the game needs to be able to display animation at around 60 frames per second. Fulfilling this requirement implies that the `render` method that draws each frame to the display should take no more than 16 milliseconds.

Sue decides to write a simple CodeAware *probe* attached to an actor that notifies her every time a performance decrease is detected in the *render* method. She starts by selecting the artifact she wants to monitor:

```
artifact 'render_func' (function: GameEngine::render)
```

In this example, the artifact is a method (or function), which is rather fine grained. One could, for instance, define a coarser artifact, such as a class, using the `class` keyword:

```
artifact 'game_engine_class' (class: GameEngine)
```

After declaring the artifact, Sue creates a probe that will profile the artifact (a type of dynamic analysis) every time a new version of the method is committed to the shared repository. The profiler will instrument the code and run the project's test suite multiple times. The probe's signal will equal the maximum execution time of the artifact.

```
performance_probe 'render_probe' (
  on: scm_commit,
  run: test_suite,
  signal: filter(max))
```

The probe may now be attached to the artifact:

```
'render_probe' -> 'render_func'
```

Sue wants to be notified via email, so she declares an email notifier, a specialized actor:

```
email_notifier 'my_notifier' (address: 'sue@xyz.com')
```

Now, Sue needs to define a coordinator that listens to the probe's signal. If the signal strength is above the set threshold of 16 milliseconds, the coordinator actuates the email notifier to send an email to Sue:

```
('render_probe' > 16ms) -> 'my_notifier'(
  subject: 'render frame rate below 60FPS!')
```

She is finished. On Sue's next commit, these actors will be deployed to the codebase along with any other changes. Then, CodeAware will able to immediately bring to her attention whenever a particular commit by anyone else on the team causes her game to run too slowly.

Note that this type of capability is not natively and readily supported in CI tools such as Jenkins or plugins available for them. However, it can be incorporated into CI tools through a specialized engine that is inserted into the build pipeline. The next section describes how this can be architecturally accomplished and the associated implementation challenges.

III. Realizing CodeAware

A. Concepts and Elements

CodeAware defines a rich landscape of elements that emulate a physical sensor-actuator metaphor and extend that metaphor with a logical overlay. These elements are:

a) ***Ecosystem:*** a distributed system of managed artifacts and agents. Agents can be probes, meta-probes, coordinators, actors, or dashboards.

b) ***Artifact:*** a host (monitored) subsystem that is a piece of software (local or crosscutting), such as a variable, statement, arbitrary code snippet, block, method, class, statement, file, package, subsystem, interface, API, data type, aspect, system, service, or application;

c) ***Static Probe:*** a passive agent *attached* to an artifact that can detect changes to an artifact and can perform different types of static analyses;

d) ***Dynamic Probe:*** a passive agent *injected* into an executable artifact that monitors its run-time behavior and can perform different types of dynamic analysis on that artifact;

e) ***Meta-Probe:*** a probe that aggregates input from multiple probes, possibly associated with multiple artifacts through defined aggregation operations;

f) ***Coordinator:*** an active agent that listens on probes, collects data from probes and manages actors.

g) ***Actor:*** an agent that takes action on behalf of a coordinator in response to a combination of probe inputs; for example, an actor may send alerts, update a piece of software, log an issue report, make a recommendation, or roll back a repository action.

h) ***Filter:*** a construct that transforms or filters the signal of a probe.

i) ***Dashboard:*** a passive agent with a UI for monitoring probe input and actor behavior for its subscribers.

j) ***Subscriber:*** A user or another external system that registers an interest in receiving information from certain actors or dashboards.

k) ***Notifier:*** An actor that notifies interested subscribers of an event that is of interest to them.

l) ***Recommender:*** an actor that makes a recommendation to interested subscribers (e.g., if a subscriber register interest for input from a specific actor, then a recommender may suggest that the subscriber also follow other related actors).

m) ***Updater:*** an actor that modifies an artifact.

n) ***Reporter:*** an actor that reports an event to a target external system, e.g., logs an issue in an issue tracking system, generates a requirement to add to a requirements management system, or generates a static analysis report for the engineers to view.

To support the above, we envisage an overlay domain-specific language (DSL) with generalized constructs and a hierarchy of increasingly specialized and expressive DSLs. At the second level, the hierarchy consists of DSLs to describe first-order elements (probes, meta-probes, actors, coordinators, and dashboards). Then at the next level these DSLs are further supported by sub-DSLs that express second-order elements with predefined, but extensible, prototype libraries (dynamic and static probes of different kinds, notifiers, recommenders, updaters, and reporters).

B. Architecture

A two-prong architecture for CodeAware is illustrated in Figure 3. On the client side, engineers define CodeAware constructs and elements locally and then deploy them first to their local version of the codebase. This is the Client-Side Engine, which may live inside an IDE as a plugin. The CodeAware client environment consists of an interpreter (or interpreters for multiple DSLs) and prototype libraries. The interpreter allows the engineer to define new prototypes or reuse existing prototypes by cloning them from the libraries, define artifacts (hosts), attach probes to hosts, deploy actors and coordinators to the local codebase, and connect these locally deployed agents with each other in the codebase.

Once the changes are committed the shared repository, the Server-Side Engine takes over. The Server-Side Engine sits in the build pipeline on the CI server. The Dispatcher first separates probes, actors, and coordinators. The probes are optimized by Optimizer so that their signals are generated in an efficient manner (e.g., common analyses of multiple probes attached to the same host are executed only once) by the Signal Generator. The Signal Generator verifies that the triggers of the probes (e.g., a change in the host artifact) are satisfied and the corresponding analyses are performed (often by running external plugins, e.g. static analyzers or profilers). The output of these analyses are consolidated in a signal table. The Dispatcher feeds the coordinators to an Orchestrator, which serializes them to resolve any dependencies. The Dispatcher also forwards the actors to a Processor, which executes the serialized coordinators using inputs from the signal table, actuating the proper actors as specified by the coordinators. The desired effects are thus achieved, with designated engineers receiving notifications, issue trackers and dashboards (not shown) being updated, and reports being generated and sent to subscribers who have registered an interest in them.

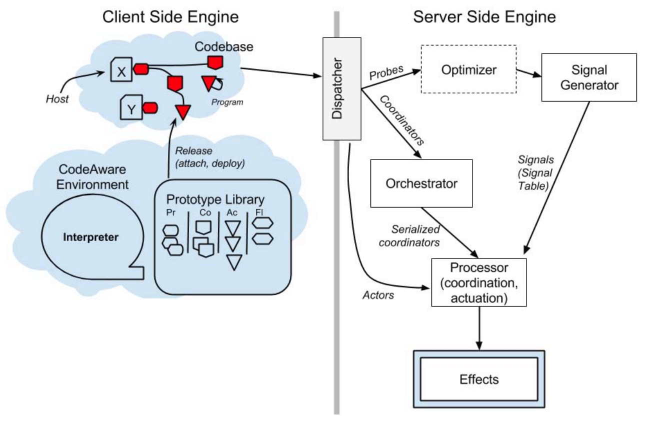

Fig. 3. CodeAware Architecture

C. Challenges

Building a system such as CodeAware poses a number of research challenges. We have identified the following as the main challenges that need to be addressed: (i) ensuring smooth evolution and continuous integrity; (ii) ensuring performance; (iii) controlling impact on sensed artifacts (footprint, performance, behavior); (iv) management and maintainability throughout the project lifespan in which artifacts, probes and actors are defined and deployed by different people (we envision specialized coordinators will have network management responsibilities); (v) defining and ensuring privacy and security (how to distinguish a user action from an actuator action, how to handle artifact access privileges for probes); (vi) interoperability of agents; (vii) dealing with code evolution (what happens when the code is refactored, how to maintain the binding between hosts and probes as hosts evolve and change identity); (viii) how to ensure scalability (what happens when there are thousands of probes scattered around); (ix) how to ensure that the system remains extensible and continues to support incremental evolution.

The proposed ecosystem unifies both a programmatic paradigm and a physical metaphor with existing analysis techniques and quality methods. It crosscuts many research fields, including DSLs, sensor networks, static analysis, profiling, continuous integration, recommender systems, technical debt, software entropy, mining software repositories, defect prediction, issue tracking, and process instrumentation. As such, the approach builds upon these existing research areas. Because of its distributed nature, the approach is particularly applicable in globally distributed software engineering, where tool support is critical [3].

IV. Comparison to Related Work

The system closest to the approach proposed in this paper is Hackystat [6], an open-source software telemetry framework for automated collection and analysis of software engineering process and product data. Similar to CodeAware, Hackystat adopts an in-process and unobtrusive approach. However CodeAware's distributed nature differs from Hackystat's local approach: Hackystat collects data directly from engineer's activities and local artifacts inside the development environment, and then may aggregate these data for visualization by the team. Hackystat sensors are associated with the local project within the local development environment, and not with the shared codebase and its artifacts. CodeAware stays loyal to the physical sensor-actuator paradigm in that the agents, sensors and actuators move with the code, rather than being confined to a single local development environment. Hackystat is a passive system, and unlike CodeAware, does not define agents that perform corrective and preventive actions.

V. Concluding Remarks

This paper proposed CodeAware, a sensor-actuator-based ecosystem for fine-grained monitoring and management of software artifacts. The ecosystem does not only provide integrated mechanism for giving early and targeted feedback to engineers about parts of the code that are of interest to them on an individual basis, but also allows engineers to automate follow-up actions. CodeAware represents our future vision of CI systems. We believe that CI systems should evolve to consider developer and team perspectives simultaneously and unify them.

Our colleague Dr. Burak Turhan's following analogy best expresses the paradigm shift underlying CodeAware:

When you want to instrument a car, you don't attach sensors to the factory, but you attach them to the car.

References

[1] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the 29th International Conference on Automated Software Engineering*, pages 55–66, 2014.

[2] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *Companion to the 21st Symposium on Object-oriented programming systems, languages, and applications*, pages 673–674, 2006.

[3] Kevin Dullemond, Ben van Gameren, and Rini van Solingen. How technological support can enable advantages of agile software development in a GSE setting. In *Proceedings of the 4th International Conference on Global Software Engineering*, pages 143–152, 2009.

[4] Martin Fowler and Matthew Foemmel. Continuous Integration. *(ThoughtWorks) http://www.thoughtworks.com/ContinuousIntegration.pdf*, 2006.

[5] Reid Holmes and Robert J. Walker. Customized awareness: Recommending relevant external change events. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 465–474, New York, NY, USA, 2010. ACM.

[6] Philip M Johnson, Hongbing Kou, Joy M Agustin, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 136–144, 2004.

[7] Thomas Kowark and Hasso Plattner. Analyzed: a shared tool for analyzing virtual team collaboration in classroom software engineering projects. In *The 2012 International Conference on Frontiers in Education: Computer Science and Computer Engineering*, 2012.

[8] Arie van Deursen, Ali Mesbah, Bas Cornelissen, Andy Zaidman, Martin Pinzger, and Anja Guzzi. Adinda: a knowledgeable, browser-based IDE. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 203–206, 2010.

[9] Andreas Zeller. The future of programming environments: Integration, synergy, and assistance. In *Future of Software Engineering*, pages 316–325. IEEE Computer Society, 2007.