November 2021

GENERATIVE SOFTWARE ARCHITECTURE ORACLE

Sam Lin

Larry Sun

GENERATIVE SOFTWARE ARCHITECTURE ORACLE

ABSTRACT

A system may assist the development-to-deployment workflow by presenting information about the architecture of a software system in real-time to facilitate understanding of the architecture. The system may analyze metadata collected from the software system to extract information about the code dependency and performance of at least a portion of the software system from end-to-end. In some examples, the system may predict the results of software testing based on the extracted information, which may help with identifying redundant testing that can be omitted from the development-to-deployment workflow.

DESCRIPTION

In general, developers may need to understand a significant portion of a software system to successfully create a marketable product. However, software systems may be extremely large and complex. For instance, a codebase of a software system may include millions, if not hundreds of millions, of lines of code. Furthermore, various aspects of the software systems (e.g., the size of functionality, lines of code, events, edge of software, legacy code, etc.) may be constantly changing and growing. Thus, because of the enormous and evolving codebase of many software systems, developers may find it difficult to learn about software systems in a time-efficient manner (e.g., such that knowledge acquired about a software system does not quickly become outdated due to frequent revisions to the software system).

During the software development process, a development team may need to perform peer review. As the development team may have various units that each concentrate on distinct portions of the architecture of the software system, the units may not be as familiar with the

portions of the architecture for which they are not directly responsible. As a result, the units may not be able to easily foresee how problems may propagate to other subsystems or adjacent systems because of the complexity of the architecture as a whole. In addition, testing for errors may be so time-consuming as to be impracticable or otherwise inconvenient. For example, the tests may be too granular or, on the other hand, too abstract to provide meaningful information to software developers.

In accordance with techniques of this disclosure, and as shown in FIG. 1 below, a system 100 may assist the development-to-deployment workflow by presenting information about the architecture of a software system in real-time to facilitate navigation of the architecture. System 100 may analyze metadata collected from the software system to extract information about the performance of at least a portion of the software system from end-to-end. In some examples, system 100 may predict the results of software testing based on the extracted information, which may help with identifying redundant testing that can be omitted from the development-to-deployment workflow.

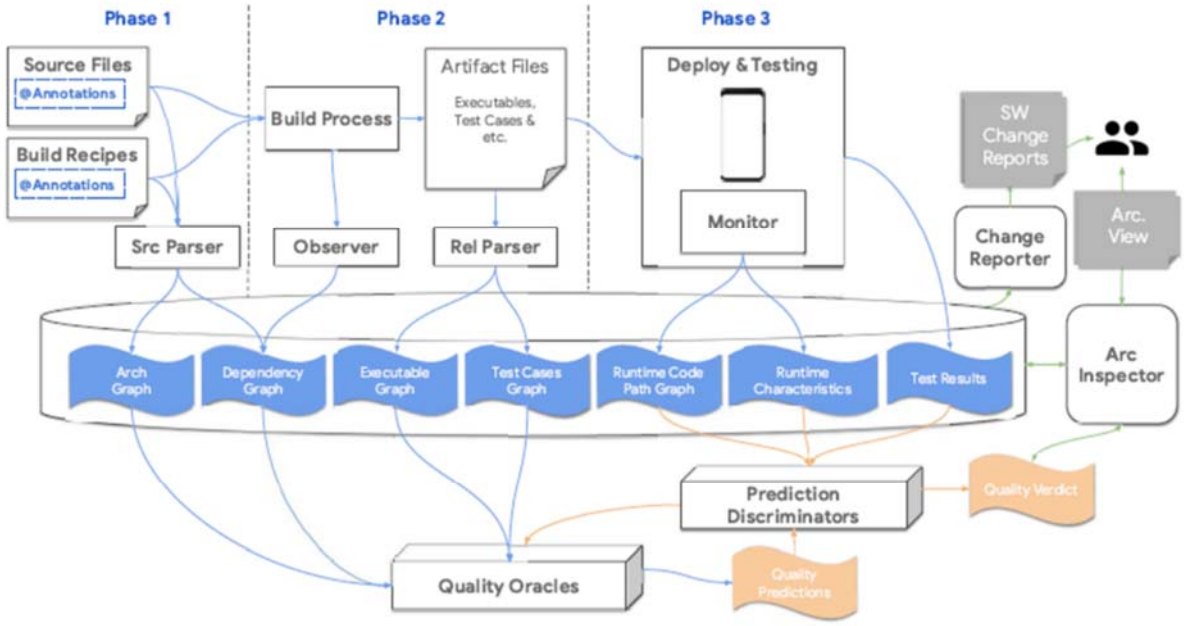

FIG. 1

FIG. 1 is a conceptual diagram illustrating system 100. As shown in FIG. 1, system 100 may include components addressing each phase of the software development cycle. The software development cycle may be divided into three phases: (1) a source code metadata set phase, (2) an artifact structure metadata set phase, and (3) a runtime metadata set phase. In each of these phases, system 100 may generate one or more metadata sets in the form of graphs (e.g., a graphical element) that include nodes (e.g., points representing objects, entities, etc.) and edges (e.g., connections between two nodes indicating dependencies, interrelationships, etc.). For example, in the source code metadata set phase, system 100 may generate an arch graph and a dependency graph. In the artifact structure metadata set phase, system 100 may generate an object dependency graph, an executable graph, and a test cases graph. In the runtime metadata

set phase, system 100 may generate a runtime code path graph.

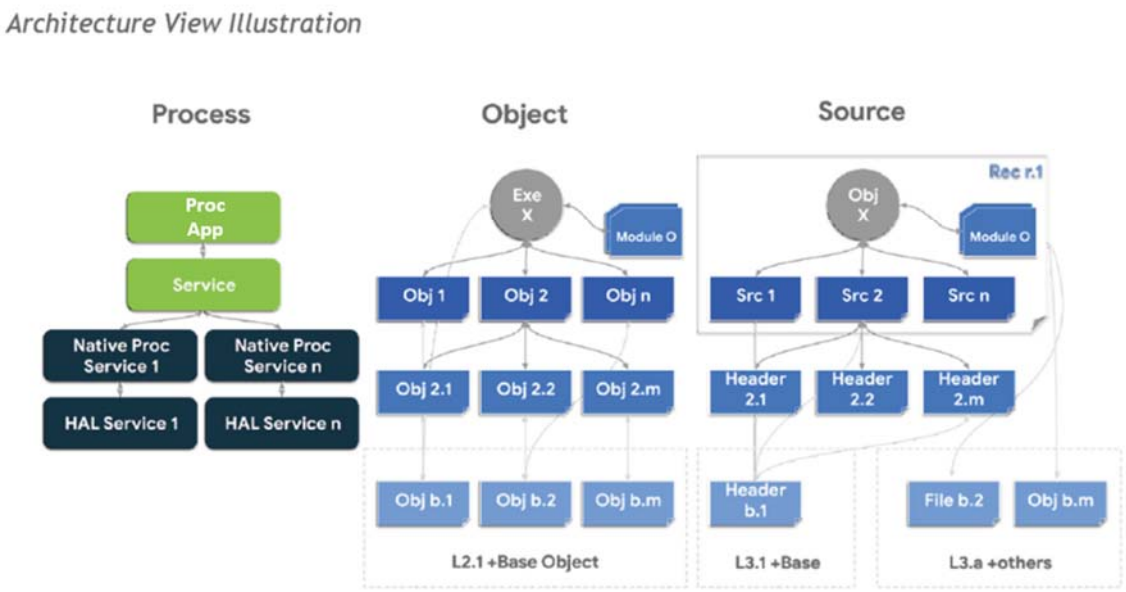

FIG. 2

FIG. 2 is a conceptual diagram illustrating an architecture graph. System 100 may generate the architecture graph in the source code metadata phase. System 100 may also generate a dependency graph in the source code metadata phase. In some examples, system 100 may generate the architecture graph and dependency graph based on metadata and referenced interfaces extracted from the source files and the build recipe. For instance, system 100 may annotate the architecture of the software system and include the annotation in the source files and the build recipe (e.g., computer-readable instructions for transforming source code into binary). A source parser of system 100 may extract (e.g., during code authoring, change submissions, post-processing for a particular revision of source repository, etc.) metadata and referenced interfaces from the source files and the build recipe based on the annotations. The source parser may extract the metadata in real-time.

The architecture graph may describe key code structure and interrelationships between nodes. The nodes of the architecture graph may represent modules, packages, classes, source files, etc. The edges of the architecture graph may represent interrelationships of connected nodes via any kind of software interface (e.g., languages, codes, and/or messages that programs use to communicate with each other and to hardware), such as an application programming interface (API) interface. The dependency graph may describe how source files are dependent upon each other (e.g., based on the annotations). The nodes of the dependency graph may represent source files or logical modules. The edges of the dependency graph may represent dependencies of the source files. In some examples, the dependency graph may indicate the size, number of lines of code, interface definitions, access controls (e.g., permissions), content hash, etc., of each node of the dependency graph. In some examples, the dependency graph may indicate the source files that each node of the dependency graph is configured to include or import, as well as the interfaces that each node is configured to reference.

An architecture inspector of system 100 may enable a user to visualize and interact with the dependencies of the architecture by different views. In some examples, the architecture inspector may generate a visual representation of the architecture graph shown in FIG. 2. A user may navigate between the different views to inspect a software stack, and each view can have multiple levels of scope and detail. With respect to FIG. 2, the architecture inspector may help a user learn more about various aspects of the processes (e.g., runtime, dynamic, execution structure, etc.), objects (e.g., build-time, static, deployment structure, etc.), sources (e.g., code-time, static, development structure, etc.), and the software system (e.g., all active processes and resource relationships at runtime snapshots, which can be filtered based on preference). The architecture inspector may also support additional annotated overlays to illustrate a system in

actions, such as software changes (e.g., architecture and dependencies changed by a patch and between two revisions), runtime (e.g., test case execution code paths and runtime characteristic statistics), and dataflow (e.g., a test case execution data flow with integrated data-flow analysis).

In the artifact structure metadata set phase (e.g., after the source code is compiled), system 100 may generate an object dependency graph, executable graph, and test cases. System 100 may generate the object dependency graph, executable graph, and test cases graph based on metadata from the artifact files (e.g., a binary artifact) created during the build process. In some examples, the metadata may indicate the actual (static) dependencies of the artifact files (which may differ to some extent from the dependencies described by the dependency graph). A relationship parser of system 100 may extract the metadata from the artifact files.

The object dependency graph may describe dependencies not described by the dependency graph of the source code metadata set phase. For example, the object dependency graph may describe the runtime dependency and other low-level dependencies. Conversely, the object dependency graph may not describe dependencies described by the dependency graph (e.g., because those dependencies were not actually used during the build process). In any case, the nodes of the object dependency graph may represent object files or logical modules. The edges of the object dependency graph may represent dependencies of the object files.

In some examples, the object dependency graph may indicate the size, content hash, etc., of each node of the object dependency graph. In some examples, the object dependency graph may indicate dependencies, such as referenced interfaces (e.g., a link to an API stub library), links (e.g., a static link, a build time link, a runtime link, etc.), and so on of each node. In some examples, system 100 may append the object dependency graph to the dependency graph.

The executable graph may describe the actual code path dependencies of the binary artifact. The nodes of the executable graph may represent executable files. The edges of the executable graph may represent inter-process communications. In some examples, the executable graph may indicate the size, content hash, etc., of each node of the executable graph. In some examples, the executable graph may indicate binder calls, sockets, pipes, queues, shared resources (e.g., memory, files, etc.), and the like of each node.

The test cases graph may describe the respective portion of a codebase that each test case uses (e.g., because a change to a portion of the codebase is more likely to affect a test case using that portion than a test case that does not). The nodes of the test cases graph may represent test cases. The edges of the test case graph may represent APIs under test and requirements under test. In some examples, the test cases graph may indicate the test module associated with each node of the executable graph. In some examples, the test cases graph may indicate the API under test, the requirement under test, and the like associated with each node.

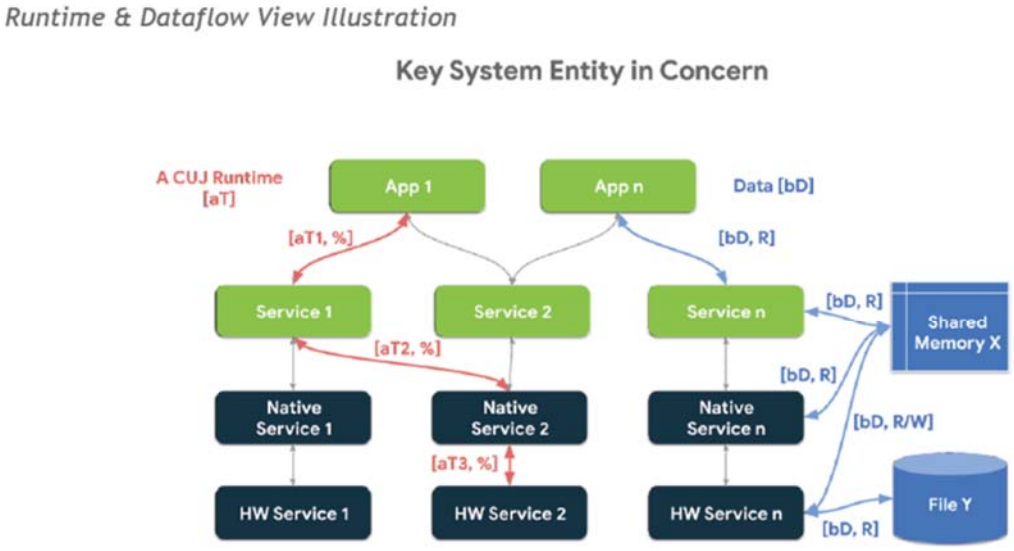

FIG. 3

FIG. 3 is a conceptual diagram illustrating a runtime code path graph. System 100 may generate the runtime code path graph in the runtime metadata set phase (e.g., during deployment and testing). In the runtime metadata set phase, system 100 may collect the results of testing (e.g., pass or fail, key test measures, retry counts, assertions, performance measures, etc.). System 100 may collect runtime code path signals to generate the runtime code path graph. The runtime code path graph may describe dependencies used during runtime. Runtime dependencies may be different than dependencies determined from source code labels and/or static dependencies (e.g., determined from the binary artifact) because of the runtime environment, software techniques that do not involve dependency injection, such as dynamic linking, etc. System 100 may also collect runtime characteristic signals indicating the performance capabilities of each test run. In some examples, system 100 may extract information from the runtime characteristic signals explaining the difference in performance of various devices (e.g., why test code performs better on a first device than on a test device).

A quality oracle of system 100 may predict test results based on information collected during each phase of software development. For example, as shown in FIG. 1, the quality oracle, which may be a trained machine-learning model, may receive information from the architecture graph, dependency graph, executable graph, test cases graph, etc., and output a prediction of the results of testing (e.g., a probability that the test result will be a pass). In some examples, the quality oracle may output a prediction in real-time (e.g., within a short period after the changes are made). In this way, the quality oracle may provide quantitative data to a user of system 100 to help with measuring work product quality throughout the development-to-deployment workflow. The predictions may indicate how changes to the codebase of a software system

affects various software quality dimensions. The quality dimensions may include, but are not limited to, affected test cases, functional regression risks, and performance regression risks.

The predictions outputted by the quality oracle may feed into a prediction discriminator. The prediction discriminator, which may be a trained machine-learning model, may generate training data (e.g., labeled data) for the quality oracle by changing properties of the runtime environment, which the quality oracle then evaluates. Accordingly, the quality oracle and the prediction discriminator may create a feedback loop in which the quality oracle and the prediction discriminator train each other, potentially leading to more accurate predictions by the quality oracle.

System 100 may generate a change report that includes various dashboards and reports to help a user of system 100 measure various quality dimensions during the software development cycle. The change report may include statistics such as artifact statistics (e.g., file size, package size, module size, lines of code, etc.), module grouping, module dependency statistics, module coupling scores, changes for multiple releases or branches, and so on. A user may also define statistics that the change report can include. Change reports may be used for code review, hidden dependency warnings, design critiques, inspections, release notes, refactoring, rearchitecting recommendations, porting, upgrade planning, etc.

One or more advantages of the techniques include facilitating the creation of vendor components that deliver device data to application developers. Such vendor components may transform software development by addressing common pain points in the industry. For example, vendor components in accordance with the techniques may flatten the learning curve for navigating and understanding the codebase of a software system. In this way, the techniques may

assist the training and improve the performance of software developers, particularly when they are relatively inexperienced, thereby increasing their productivity.

It is noted that the techniques of this disclosure may be combined with any other suitable technique or combination of techniques. As one example, the techniques of this disclosure may be combined with the techniques described in U.S. Patent Application Publication No. 2013/0305223A1. In another example, the techniques of this disclosure may be combined with the techniques described in U.S. Patent Application Publication No. 2018/0210709A1. In yet another example, the techniques of this disclosure may be combined with the techniques described in "What is Software Development Life Cycle Management (SDLC), and what are best practices, tools and templates for teams and organizations?," Praxie, October 20, 2020. In yet another example, the techniques of this disclosure may be combined with the techniques described in "SDLC – The Software Development Life Cycle," Project Manager, October 20, 2020.