Harness

A State Testing Environment for Big Data Algorithms

White Paper

Ryan Berkheimer
Global Science & Technology, Inc.
NOAA's National Centers for Environmental Information
Asheville, NC

Abstract

Harness is a generic environment designed to handle easy loading, parsing, and evaluation of complex software projects and project groups. Large-scale systems consisting of multiple languages and intricate computational routines can be loaded into Harness as components and parsed into internal and external functions that can in turn be combined into workflows to be evaluated both in execution metrics and workflow state. Evaluations with matching output types can be compared against each other for differences as projects evolve over time.

Introduction

Agencies that handle immense scientific datasets, such as NOAA's National Centers for Environmental Information, build and maintain numerous modeling and data manipulation software projects. Each project typically involves multiple distinct parts consisting of programs, scripts, and data. These parts can represent decades of work by multiple authors across a diverse set of programming languages, technologies, and styles.

In the scientific domain, particularly at an agency with authoritative mandate, software projects are designed to reproduce results of peer-reviewed research. As research methods of a particular scientific algorithm or model change over time, any associated software is modified to maintain the reproducibility of the updated results. Each software change results in increasing complexity, leading to an increased number of possible project states. In software, the number of possible states is highly and positively correlated with natural degradation in testing coverage, documentation, and overall quality. Plainly, the probability of a project becoming inoperable—too obtuse to use, producing unexpected undesirable results, or simply breaking—grows with every project change.

Once a project is broken, there is an enormous associated cost in returning it to operability. To mitigate this risk, software must be developed in a manner that resists the symptoms of natural degradation. Documentation, record of change, and comprehensive testing coverage must be maintained for projects that are written in an arbitrary set of languages, over long periods of time, by a diverse and revolving set of contributors.

While there are many options to address these challenges individually, there are problems associated with applying disjointed systems, including the absence of desired functionality related to testing evolution and persistence over time. Harness attempts to present a unified environment for development and testing that grows naturally, is self-documenting, and provides intuitive and efficient ways to maintain and grow test coverage.

Design Overview

At its top level, a Harness deployment is an empty collection of harnesses, as any number of Harness projects can exist inside a Harness instance. Inside a particular Harness, any number of existing user projects (the scientific algorithm, model, and data projects being developed) can be added as components. Components consist of individual or grouped scripts, algorithms, Makefiles, and datasets, among others. A component is loaded either from a file system location or a version control system and is stored locally on the system disk, so that it can be used in execution, as well as in the harness component collection in database storage.

Components contain derived functions of both internal and external type. An internal function is a one-to-one mapping of a callable subroutine, function, or module within a particular component. An external function is something that calls the component as a whole (a shell command that calls a compiled program, for example). Functions are not stored with their parent components, but in their own function collection in the harness container. This storage pattern allows functions from any component in the Harness to be chained together, creating complex workflows for testing and output. Both workflows and their evaluation outputs are always stored, available at any time for retrieval and use.

To allow chaining and evaluating of functions and workflows, each one has its own set of input and output parameters, and each of these parameters is mapped to a global parameter definition. The parameter definitions are what allow functions to translate between each other, matching outputs to inputs. Any function or workflow can be evaluated by providing necessary inputs. All evaluations are stored in a separate collection, containing evaluation metadata such as run time and resource usage, the inputs that were used, and the output values that were produced. Evaluations with matching outputs, based on their parameter definitions, can be compared. Comparisons are then stored similarly to evaluations.

Design Tenets

Functional Workflows

A harness consists of components, and components consist of functions, both internal (derived from the component code) and external (applied to the component via some external call). Functions can be chained together into workflows, or compound functions, when output from one function matches an input of a second function (based on parameter definition). Each function and workflow automatically maintains a current set of input, intermediate, and output parameters when updated. Workflows can be evaluated by simply providing the necessary inputs. The resulting evaluation is saved in storage for later use. Function and workflow storage is designed so that they can be evaluated as they were at any point in their modification history.

Persistent Results

Because state testing needs temporal coverage, workflows and evaluations are saved as they are created or modified so that they can be compared and analyzed against future iterations. Additionally, as projects evolve, their parts change unpredictably. To account for this unpredictable change while simultaneously providing both self-documentation and the ability to recall and evaluate any previous system state, all parts of a Harness (components, functions, evaluations, parameters, and others) save every permutation so that all previous system states are available to be recreated exactly. Harness uses both database and local storage to maintain the ability to recreate these states.

Version Control Support

Harness components can be directly added from a VCS repository. Once added, both a component and its VCS have the ability to be enabled or disabled. A component with an enabled version control system will keep itself updated with the latest version, but will allow access to previous version states. When a component is updated, its record is duplicated into a new database record and the existing functions are copied to the new record. Currently git and a rudimentary file system source are currently supported with plans to add support for other VCS systems.

Collaboration

Harness is designed so that many users can be hooked into the same harness system, with permissions assignable to various harnesses at a macro and micro level. This allows multiple users testing in the same environment, creating and sharing workflows and evaluations with their team. Harness is currently designed as a single deployment system with plans to add support for a distributed system in order to leverage individual system resources while sharing database components.

Utility

While the Harness project includes a built-in web-based front end, Harness' primary utility is as an engine with a REST API. It provides access to all of its methods as micro services, which can be combined or consumed in arbitrary ways. This design strategy supports more open-ended uses for the Harness project, including custom UI or using Harness as a simple means to expose legacy projects as web-based APIs for other projects.

Compatibility

Harness is designed with simple and universal technologies in mind, so it can be put on systems with stringent IT security.

Harness is written in the interpreted python language with wide acceptance and distribution of the necessary interpreter. Additionally, Harness contains all its dependencies. Harness currently supports Mongo storage, with plans to add RDBMS support for systems with even more stringent security concerns.

The project codebase itself is small. Its current storage architecture is designed so that it will not conflict with anything else that might be using the chosen storage system.

As a service oriented python project, Harness does not need a container like Tomcat to deploy – it automatically deploys on localhost.

Open Source

Harness is developed at NOAA NCEI in Asheville, NC on a contract by staff employed by Global Science & Technology, Inc. As such, it falls under public domain under the MIT license. As an open source project, collaborative development is encouraged.

Architecture

Harness uses no explicit classes or objects, and state is handled through python dictionaries, with models built and defined by a generic named tuple to dictionary system. This way, custom data types used by components can be defined in situ. Its coding style also emphasizes functional programming methods when possible, where functions are handled and returned within closures, to be used as first class objects. This design emphasizes a clearly defined scope, and results in both high utility and code clarity.

Harness was developed as a model-view-controller structured project. Because it is structured as an MVC, bidirectional logic flows naturally between controllers, processors, and data access objects.

Rules that govern how project layers can communicate help guide development. For example, processors can only talk to other processors that are contained by themselves or at the same level as themselves, and a processor can only talk to its counterpart DAO and controller methods. Each program layer handles only things related to what that program layer should do—for example, DAOs handle only data access and never handle any data manipulation. This style allows a small team to develop the complex and growing Harness project.

The current mongo back end easily allows for a malleable schema. As Harness code changes and features are added, they can be tested and incorporated easily because of the way Mongo uses flexible document schema rules within collections.

Harness is designed with a source database that maintains references to all projects existing in the instance and a separate database for each harness. This structure maintains project separation and good horizontal scalability across systems and distributions.

Case Study

Developed by several contributors over many years, the Pairwise Homogeneity Algorithm (PHA) is an algorithm created by meteorologists and data scientists at NOAA's NCEI Asheville, with the primary goal of detecting and removing faulty field instrument response functions from long-term historical climate data in temperature datasets. In recent years, it was decided that the project source code needed reengineering. At the start of refactor, the primary language was Fortran 77. There were several thousand lines of code, hundreds of GOTO statements, dozens of global variables, and relatively few subroutines. Many subroutines were several hundred lines of code. The algorithm relied on dozens of associated scripts.

Each script was written either as a makefile, a bash script, an awk script, or a gawk script. Some paths in the scripts were hardcoded to a particular distribution. The project was designed to run on two distinct station sets, with each station set containing three types of temperature data. Station data also has separate history files. Each set of data was stored in a separate tab-delimited text file. There were several versions of the project. A full run of the algorithm took approximately 4 to 6 hours per dataset. The outputs of the algorithm have been peer reviewed and published, meaning the refactor work had the additional requirement of maintaining the original results as the project was cleaned up and refactored.

Case Study cont'd.

For this project, a use case of harness could be:

1. Create a new harness called 'pha refactor'
2. Load the original code in its entirety into a folder, and add that folder to a git repository
3. Add the git repository as a new component, called 'pha algorithm'
4. Load anything necessary to run the entire algorithm from the scripts into an external function
5. Specify inputs and outputs for this external function - say a station file, a station type, an output temperature file
6. Write a simple python module to load and store the station data and create subsets of station data based on date ranges, locations, etc.
7. Add this python module as a component called 'data prep'
8. Add the internal python module functions as needed
9. Create new compound functions using the parameter map to make workflows between the 'data prep' component functions and the 'pha algorithm' functions
10. Use the workflows to run as many evaluations as the internal structure of the 'pha algorithm' as needed
11. Run comparisons between subsequent workflows
12. Use analytic tools to display the output from these evaluations and comparisons

While this is a possible use, it is useful to note that Harness does not insist on any one way of performing this development. There could be multiple copies of the PHA algorithm, each called something different, that reference different development arms. Different output components could be added or created to do different analyses of the output station temperature data, or different input programs could be written to produce different, compelling input data sets. Harness puts no constraint on the 'how' of development.

Future Work

Harness is in active development, and while development is progressing smoothly, the progress is dependent upon contractual obligations and funding availability. Harness is currently in use by developers as a testing tool. The goal is to have Harness officially come online by May 2016 for use by software test engineers, with a basic internal web based GUI operational by Summer 2016, if funding and schedule allows.

Once the project is complete, the major goals are:

- Integrate it with an RDBMS backend
- Add support for more VCS systems
- Continue to add functionality and a more comprehensive REST API
- Begin adding automatic loading of functions (both internal and external) for different languages
- Add a server construct to hold different filters, like compilers, etc.
- Disconnected distributed system, using local system resources and sharing results