INTRODUCTION

After the initial development, all software has many maintenance and evolution stages, each introducing new improvements, functionalities, or further optimizations. Those step-by-step refinements and tuning can create many feedback loops, easily jeopardizing a project's integrity and sustainability. The DevOps movement provides new data points to identify and quantify the structural changes in source code beyond the traditional line-based CHURN. Static Analysis can extend DevOps principles to offer clear, granular, and actionable views of "what changed between versions of source code," enabling new strategies that let developers maintain better control over the evolution of any software project. The mid-term result is less costly rewrites and project cancelations, and benefits are even more significant when your team relies on extensive cooperation with 3rd party teams and dependencies.

What this plugin is about: This plugin is a framework for scripting & managing "comparison" and "changes between versions" with SciTools Understand Python API. It will provide the following:

- comparison.py: a "Loader" class to create **call trees that include change information**.
- architectureparser.py: an "ArchitectureParser" class that lets you specify "scenarios" (more later about this) using Architectures in Understand.
- pather.py: a "Pather" class to help **determine paths (and related changes in them) starting from one set of functions to another.**
- GraphComparison.upy: A plugin Graph to display the output of comparison.py
- GraphPathViewer.upy: A plugin Graph to show the result of pather.py
- ComparisonCodeCheck.upy**: A plugin Codecheck checker that highlights risky changes in your project** based on pather.py

How to install: install the project using install or install.bat

USING IN THE GUI

How to get started:

- Open an Understand Project
- Make sure you have a "Comparison Project" set (under Compare >> Comparison Project)

To use the plugins:

USE CASE 1: Display a "call" tree (or "call by" tree) and the changes in it.

(It is very similar to the default provided call tree, except it will allow you to limit the display to what has changed)

- Select a function
- In the context menu, run "Graphical Views >> COMPARISON Graph."
- It will show the changes in the "call" or "call by" tree for this function

Your options:

- Depth [1-7]: The Depth by which the computation is happening [1-7]. The deepest, the more calculation is needed.
- Display Change Only [on/off]: Only show the path with "changes" in them. The graph detects change only at the specified Depth. If you select a Depth of 2 and a change happens at a depth of 3, the "Display Change Only" version of the graph will not show this change.
- Mode [Call/Call By]: Set the algorithm's direction for seeking functions.
- Orientation: Orientation of the graph.

READ BEFORE USE CASE 2 and USE CASE 3: SCENARIO(s)

The subsequent two use cases (Critical Path Graph and Critical Path) provide information based on scenarios. A Scenario is your ability to specify paths (call tree) by where you want to start and where you want to end up.

Example(s) of scenarios:

Scenario A: all path(s) from main() to memcpy()
Scenario B:all path(s) from any functions that use std::stdin to any functions that use memcpy()

How does one specify a scenario in the Understand GUI?:

- Create an architecture that contains [CRITICAL] in its name (including the brackets) at ANY level
- Create directly under it a sub-architecture named "from" and another one called "to"
- Either add functions entities in them or a "specification tree" using Entity and Reference Kind name from Understand (see example). A specification tree can have multiple layers; for instance, use "function/call/function/use/A" to specify a function that calls a function that uses A.

Example(s):

Scenario A specification in architecture may look like the following:

- [CRITICAL] main to memcpy
  - from
    - main (entity)
  - to
    - memcpy (entity)

Scenario B specification in architecture may look like the following:

- [CRITICAL] use std::stdin to call memcpy
  - from
    - function
      - use
        - ✓ std::stdin (entity)
  - to
    - function
      - call
        - ✓ memcpy (entity)

Once you have set your scenario(s), you are good to use the following two use cases:

USE CASE 2: Showing Critical Path(s) based on a scenario.

The graph will display the path(s) for specific scenarios.

You can display the graph in two different ways:

- In the main menu, select “Graph >> Project Graph >> CRITICAL PATH Graph”: This will show all the path(s) for one scenario at a time (see “Your options”)
- In the context menu of a function entity, select “Graphical Views >> CRITICAL PATH Graph”: This will show all the path(s) that contains the entity for one scenario at a time (see “Your options”)

Be Patient. WARNING: These graphs can be highly computative and may take time before they show. When you change the options in “your options,” the computation will be faster as most information is cached.

Your options:

- Scenario: This lets you select the architecture used for creating the scenario. You can choose only one at a time.
- Display Changed Paths only [on/off]: You can select if you want to show only the paths that contain changes. We recommend leaving this option “on” as the graph can quickly get “gigantic” and “useless.”
- Performance Check [100,1000,10000, off]: The computation of paths always takes place but displaying the graph happens if the number of paths is less than the threshold selected. This option is safe for avoiding overloading the display engine by accident.
- Simplify Graph [on, off]: replace paths in which only a single function calls a single function with “…”
- Group Start and End: group the start and end of all path(s) together
- Break by change: unless it’s off, the plugin will show subgraphs containing subsets of the original paths:
    - Change by change: each subgraph will only show the paths for one specific change.
    - Independent: each subgraph will group independent paths (paths that do not connect between subgraphs)
    - By start: each subgraph will show the paths starting at the same point.
    - By end: each subgraph will show the paths ending at the same point.
    - By start and end: each subgraph will show the paths starting and ending at the same points.
    - Break all paths: Each path will get its subgraph.
    - off

USE CASE 3: Detecting Risky Point(s) based on a scenario.

The Codecheck Checkers will find all the entities that:

- Have changed
- Are at least in one path described by the scenarios.

To use, select “CRITICAL_PATH_CMP Critical Path Comparison” in your checkers.

Be Patient. WARNING: This check can be highly computative and may take time before results show.

Note:

- This check runs on ALL the scenarios; you cannot select only one.
- Each item is assigned a “PRIORITY.” We compute the PRIORITY as follows:

  PRIORITY = # PATHS * # SCENARIO * M

  Where

  # PATHS is the number of the changed paths the item is part of

  # SCENARIO is the number of scenarios that flag item

  # M is 1 but 2 if the item is never a start or an endpoint in any path.