Eshaan Aggarwal
eshaan.aggarwal.cse21@iitbhu.ac.in
Uttar Pradesh, India
Timezone: Indian Standard Time (+5.30 UTC)

BenchExec: Summary Tables and Filtering Improvements

GSoC Project Proposal for Software and Computational System Lab

Personal Details and Contact Information

Hi! I am Eshaan Aggarwal, a pre-final year student from the *Indian Institute of Technology (BHU) Varanasi, India*. I am an avid open-source enthusiast who loves making (and breaking) software. I was a student mentee under *Google Summer of Code '23* for the Palisadoes Foundation.

I am available on GitHub as **EshaanAgg**. Here are my details and contact information:

- **Email Address:** eshaan.aggarwal.cse21@itbhu.ac.in
- **Major:** Computer Science and Engineering
- **Degree:** Bachelor of Technology (B. Tech.)
- **Slack:** EshaanAgg
- **Time-zone:** Indian Standard Time (+5:30 UTC)
- **LinkedIn:** https://linkedin.com/in/eshaan-aggarwal/
- **Resume:** https://rxresu.me/eshaan.aggarwal.cse21/cv
- **Portfolio:** https://eshaanagg.netlify.app/
- **Postal Address:** 701, Mahagun Villa, Sector 4, Vaishali, Ghaziabad, UP (201010), India
- **Phone:** (+91) 76786 48384

Synopsis and Goals

As a part of this project, I would like to work on improving the functionalities as well as increasing the maintainability of the table generation component in BenchExec. I aim to tackle the following issues/feature requests under this project:

- Upgrade to the latest version (v6) of React Router
- Merging the two summary tables on the main page into one
- Display the various error counts in the table summary tabs and rework the filtering functionality of the HTML table component

In addition to the same, here are some additional stretch goals for the project that I would also like to complete:

- Changing the run set names to be more user-friendly
- Add support for syntax highlighting for overlay panes
- More intuitive display for command line options in case of merged run sets
- Add links to raw results to the HTML table
- Add title lines for overlays

These are the primary issues I would spend most of my time working on. I plan to undertake a large project (350 hours) for the same, spread across 18 weeks with the following distribution:

Sub Project	**Dedicated Hours**	**Dedicated Weeks**
Upgradation to React Router 6	50	3
Merging Summary Table	90	5
Displaying Error Counts and Reworking the Filtering Logic	150	7
Buffer Period and Stretch Goals	60	3
Total	**350**	**18**

As evident from the overview above, I plan to work for almost 20 hours every week. While I have definite goals for the first 15 weeks regarding the subprojects listed and explained below, the final 3 weeks have been kept as a buffer period where I would first work on completing any technical backlogs from the previous projects that might occur due to unexpected difficulties or personal reasons. If all stick to plan (or if I can complete the earlier deliverables in a shorter time), I have also added a detailed description of the following stretch goals and how much time commitment they may require:

Sub Project	**Dedicated Hours**	**Dedicated Weeks**
Changing the run set names	10	0.5
Syntax Highlighting for Overlay Panes	20	1
More Intuitive Display for Command Line Options	20	1
Adding links to raw results to HTML Table	10	0.5
Add title lines for overlays	20	1
Total	**80**	**4**

Benefits to the Project

This proposal aims to support some significant feature requests for the HTML table component and refactor the code to improve readability and maintainability. The primary benefit of this proposal lies in the fact that it tries to address the significant issues (labeled as GSOC on the GitHub repository) and other relatively smaller issues. Having contributed to the project before, I am confident that I understand the codebase, the use case, and the design practices well and can help increase the overall quality of the codebase via more documentation and testing.

Deliverables and Plan of Action

The following is a brief overview of how I would work on the different subprojects (even the stretch goals) that I plan to undertake as a part of the GSOC term. My primary aim would be to complete the first three subprojects (Upgradation of React Router, Unification of Summary Tables, and Addition of Custom Error types) and then spend the leftover time completing as many stretch goals as possible.

Upgradation to React Router 6

The reason for taking up this issue is primarily because it is a rather mechanical issue, which is not very tightly coupled with implementing the various logic in the actual React components; hence, it would be the safest and most easily testable change. Working on this issue would also give me a better feel for the rest of the project and thus improve my confidence when I aim to add/edit functionalities of the table component in the following steps.

Expected Time of Completion

3 weeks

Approach

I aim to follow the approach outlined in the official migration docs so that the migration experience is smooth and error-free. The changes that would need to be performed include:

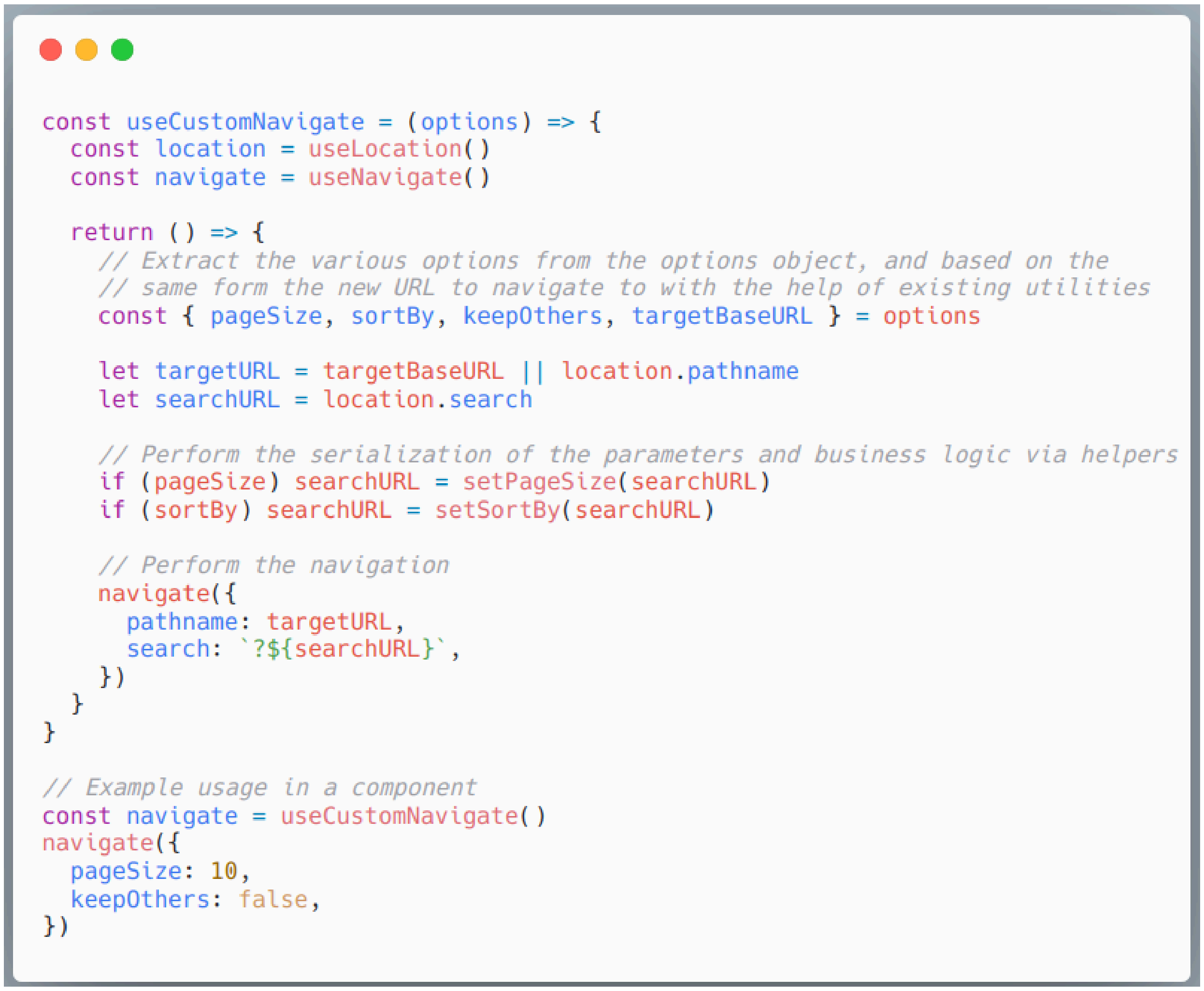
- **Replacing the <Switch/> components with the <Route/> components**

This is expected to be an easy part of the migration, where we would just be changing due to the syntax of the public API. The main changes would be present in these lines.

- **Deprecation of the useHistory hook and history object**

The source code currently uses a useEffect on the history to preserve the various URL parameters and set them correctly automatically whenever the page changes. This is not a good practice as this may result in unexpected and not easily testable logic around the handling.

Instead, we should be more explicit about the changes in the URL and the associated parameters. The most optimal way to do the same is to create a custom hook around the *useNavigate* hook provided by React Router, to which we can provide options specifying the behavior that we would like (managing the index of the pages, the query params, etc.), and the same would responsible for the redirection. A sample outline of the proposed hooks is as follows:



```
const useCustomNavigate = (options) => {
  const location = useLocation()
  const navigate = useNavigate()

  return () => {
    // Extract the various options from the options object, and based on the
    // same form the new URL to navigate to with the help of existing utilities
    const { pageSize, sortBy, keepOthers, targetBaseURL } = options

    let targetURL = targetBaseURL || location.pathname
    let searchURL = location.search

    // Perform the serialization of the parameters and business logic via helpers
    if (pageSize) searchURL = setPageSize(searchURL)
    if (sortBy) searchURL = setSortBy(searchURL)

    // Perform the navigation
    navigate({
      pathname: targetURL,
      search: `?${searchURL}`,
    })
  }
}

// Example usage in a component
const navigate = useCustomNavigate()
navigate({
  pageSize: 10,
  keepOthers: false,
})
```

This is a prospective interface for the API. I would work on designing the final one with the study of the usage of routing all over the application. This step would involve a ton of refactoring (much like the work in #1006). This hook would also require the (much-needed, in my opinion) deprecation of methods like `setHashSearch,` `setParam`, etc., which modify the location URL directly and would require the caller to make use of the singular custom hook to satisfy all it's parameter and browsing related needs, leading to enhanced readability of the code, proper documentation of how the parameters are modified, and traceability.

Unification of the Two Tables on the Summary Page

As discussed in issue #506 and other related threads, a lot of cognitive overhead is associated with having two tables on the Summary page, making for a bad user experience. This is the next issue I would like to tackle by designing a new component that can display the data from both tables simply and lucidly.

The reason for tackling this problem before any other tasks is that all other tasks would directly/indirectly involve changing the different subcomponents of either of the table components. Migrating the same to the new design would provide a solid and stable base to perform the other tasks later. If we are to tackle this task later, it might be possible that the changes we make while performing the other tasks may be such that they act as blockers for this, leading to unnecessary refactoring and effort.

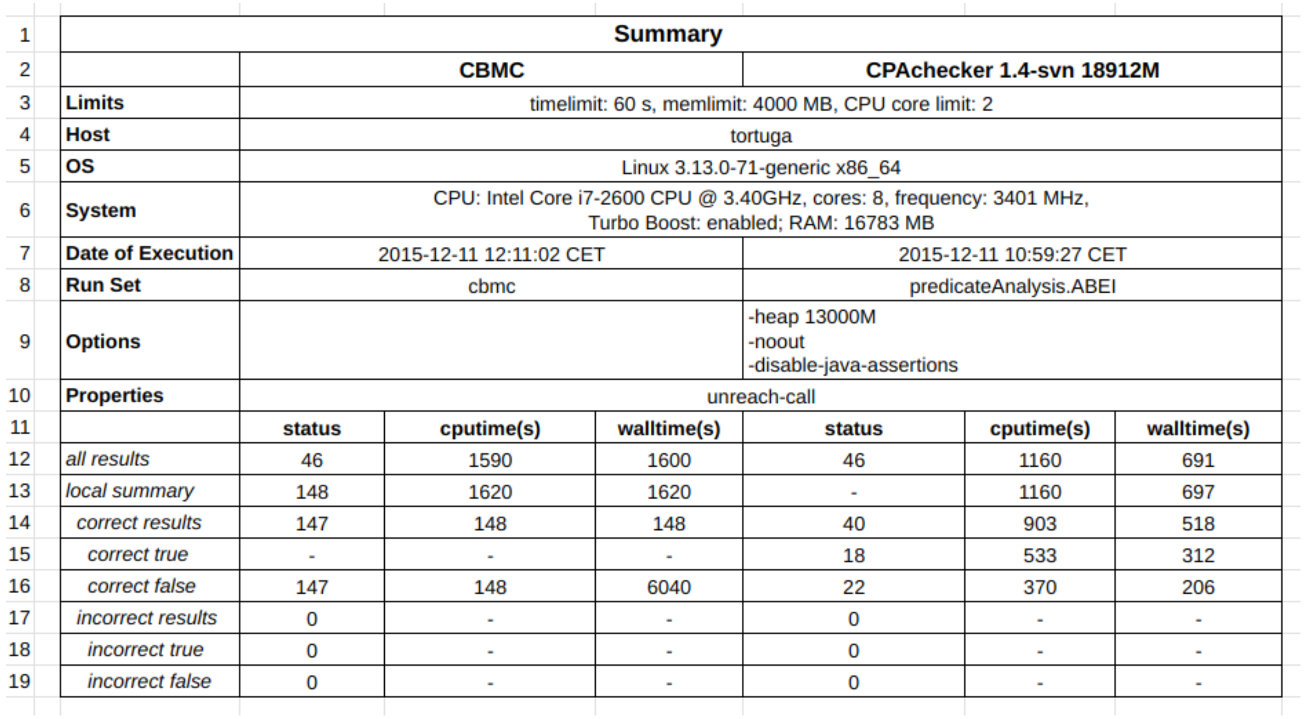
Expected Time of Completion

5 weeks

- 2 weeks: Understanding the existing functionalities and developing the initial UI
- 1.5 weeks: Adding the support for all the filters, as well as good-to-have functionalities like pinning rows
- 0.5 weeks: Adding a comprehensive test suite for all the different situations and settings
- 1 week: Integrating the new component on the main page and removing all the unused sections of the previous implementation

Approach

The goal here will be to merge the upper table on the summary tab into the header of the lower table, such that we have a multi-line header in which many cells span across different columns. I would work on developing a new component altogether (instead of refactoring the old ones). The tables shown currently on the live example would be changed to have the UI similar to the following:



Summary						
	CBMC			**CPAchecker 1.4-svn 18912M**		
Limits	timelimit: 60 s, memlimit: 4000 MB, CPU core limit: 2					
Host	tortuga					
OS	Linux 3.13.0-71-generic x86_64					
System	CPU: Intel Core i7-2600 CPU @ 3.40GHz, cores: 8, frequency: 3401 MHz, Turbo Boost: enabled; RAM: 16783 MB					
Date of Execution	2015-12-11 12:11:02 CET			2015-12-11 10:59:27 CET		
Run Set	cbmc			predicateAnalysis.ABEI		
Options				-heap 13000M -noout -disable-java-assertions		
Properties	unreach-call					
	status	**cputime(s)**	**walltime(s)**	**status**	**cputime(s)**	**walltime(s)**
all results	46	1590	1600	46	1160	691
local summary	148	1620	1620	-	1160	697
correct results	147	148	148	40	903	518
correct true	-	-	-	18	533	312
correct false	147	148	6040	22	370	206
incorrect results	0	-	-	0	-	-
incorrect true	0	-	-	0	-	-
incorrect false	0	-	-	0	-	-

Some key points regarding this component would be:

- The whole component would be implemented as a React Table (and not a manual HTML table). Although this would increase our dependency on an external package, it would provide a nicer developer experience where we would get much of the functionality out of the box (like resizeable columns, fixed headers, etc.)
- A pin icon would be prepended for the Title and information rows (rows 2-11). Clicking the icon would toggle the fixed nature of the row. This is an important feature as it helps users decide which information they want to see while scrolling the page in the case of larger data grids.
- The filtering modal will be updated in the following manners:
    - Instead of now being accessible from some text on Row 11 Column 1 (as it is currently implemented), we would append a settings icon to the right of the Header row (Row 1). Clicking the same would open the filter model.
    - The UI of the modal would now be divided into different sections, each of which can be used to group different types of settings. For this project's scope, I would work on implementing two sections:
        - The first section would allow the users to hide/unhide some information rows (Rows 3-10). This would allow them to see only the system information that they currently require. We can also work on supporting a single checkbox for hiding the header rows (all at once, making the table look similar to what is currently in the lower table). We can instead only toggle "Options," as these usually take up most of the space. The exact implementation would be decided after discussion with the mentors and feedback from the community.
        - The second section would allow the users to configure the columns they want to see (like the current implementation). Stylistic changes will be required for this section.

To implement the functionality, I would like to follow the following steps:

- I would design a new component called **HomeTable** and allow it to accept props that are the union of the props of the two individual tables right now. I would create a new page where this component would be rendered to be tested and screened. After the element is stable, we will remove and integrate this new (hidden) page on the index page.
- The most important part of the implementation would be figuring out the right number of columns (colspan) for each part of the data we wish to display. I would spend the most time figuring out and making this logic bug-free. At the end of this step, we should expect to see a very simple table, much like the one shown in the image.
- After the display is stable, I would like to work on the **Settings** panel as described in the implementation notes above. I would spend time adding tests to ensure that the table looks consistent even when a combination of filters are applied (some of the title rows and some statistic columns are hidden) and the colspan determining logic is bug-free.
- In the final stage, I would work on adding styles to the table so that the same looks more aesthetically pleasing. I would like to spend some time to ensure the styles look consistent on a variety of screen sizes, and I would also add some configuration to

customize the appearance of the table to the **Settings** panel (a switch for striped rows would be a good example of the same)

Adding support for more types in the error statistics

I would like to take up this issue next, as I feel that I would be confident enough to make changes to the project's core infrastructure by this point. I want to work on an approach that dynamically categorizes all the different status codes so that none of the unexpected codes are missed. It would also allow for visibility if there is a disproportionate number of message errors.

Expected Time of Completion

7 weeks

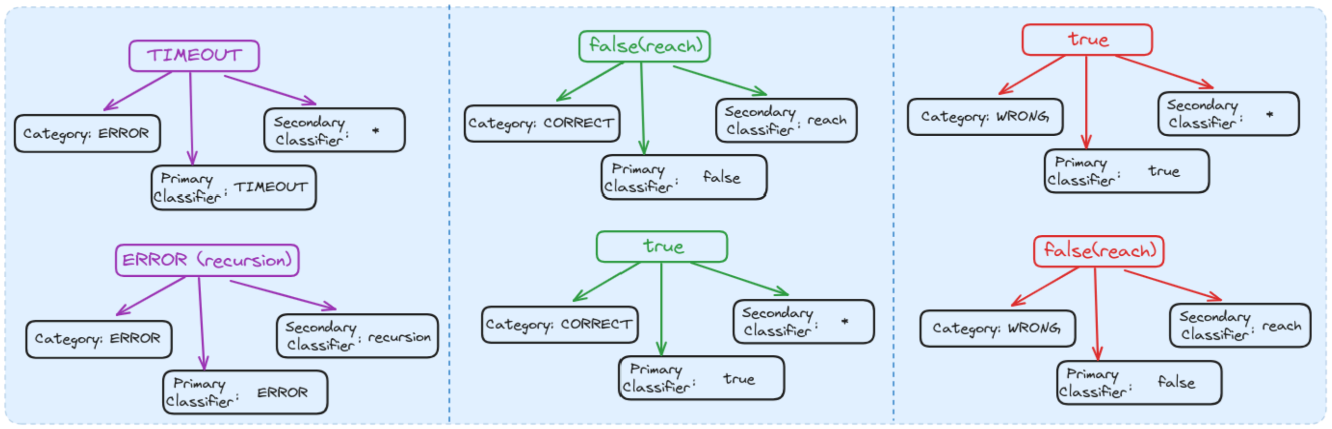
- 3 weeks: Implementation of the core logic of new error types
- 2 weeks: Summary Table UI and integration
- 2 weeks: Design the new filtering component and integration

Approach

I would like to change the display of the project to provide a three-level classification:

- Category Identifier: It indicates whether the result was CORRECT, WRONG, or an ERROR.
- Primary Classifier: In the case of CORRECT or WRONG categories, it can have the values TRUE, FALSE, or UNCONFIRMED. It could be identified by the status text of the row (like 'true,' 'false'). For the ERROR case, it would be the name of the error (like 'TIMEOUT,' 'OUT OF MEMORY').
- Secondary Classifier: This can be identified with the help of the content in the brackets of the status text (like 'recursion' in 'ERROR(recursion)' or 'reach' in 'true(reach)'). This category would contain a placeholder value (*) for all the cases that lack a bracketed subclassifier.

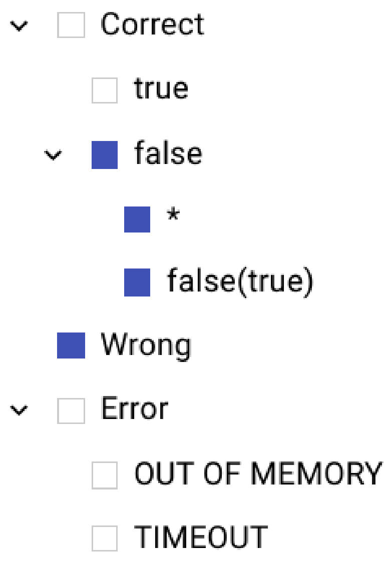
This would lead to the formation of a hierarchical clustering that would be three levels deep. Here are a few examples of how the various rows would be characterized:



This dynamic approach would characterize all the different statuses into three levels. It would assign them a default category (*) in case of absence to ensure that we can maintain that the number of parent instances is equal to the sum of the sum of the children's instances at every classification level.

The scope of this functionality can be further increased with the help of integration of the following two features:

- Custom handling for particular types of arbitrary errors like "ASSERTION." We can attach particular metadata or classify them into separate categories per the use case.
- This hierarchical clustering would also improve the UI of the filters used in the HTML table component. Instead of just listing the unique Status and Categories available, we can make the following changes:
  - Change the list to a hierarchy, with each status/category as a node. Selecting a parent node would automatically select all the child nodes.
  - Replace filters' input (radio) nature with a checkbox so that the users can select multiple nodes simultaneously.



This is a sample UI, so we can now aim for the checkbox to appear like this. We would customize the styling of the nodes according to their color (like the CORRECT nodes would be green, the WRONG nodes in red, and the ERROR nodes in pink), etc. To implement the same, we can either create the same ourselves from scratch or use pre-existing components such as React Checkbox Tree. This component should be designed to be reused for both places: in the inline filtering provided under the status column and in the filter menu, which can be accessed by clicking on the top-right of the page.

I would prefer the later approach of using a component from NPM, as implementing the same is a non-trivial task if we want to provide the end users with a good UX, which involves making the checkboxes collapsible, handling indeterminate states for checkboxes (when some of my children are selected, but not all), and having an accessible markup for the same. Using a package would abstract this complexity from us and ensure that all the functionalities related to the same are stable with a comprehensive internal test suite. Also, I advocate for React Checkbox Tree as it only depends upon React and has no other external dependencies that need to be bundled if we integrate the same into the project.

Implementing this approach would require only changes to the characterization functions in stats.js. I would also need the "statisticsRows" export to be changed to a dynamically generated function based on all of the data. Minor changes to the "StatisticsTable.js" would also be required to accommodate the changes made in the API and utility of the statistics calculation.

This component also provides an excellent opportunity to increase the project's test coverage. After the capabilities of the element are stable, I would add comprehensive tests that simulate the providing of various kinds of data to the statistics generation functions and assert that the same generates the characterization levels correctly (including for status that it has never seen before).

Here is the prospective implementation of the logic of characterization:

```
const RESULT_DEFAULT_CLASS = "*";

/*
* Classify the result of a single run into a primary and secondary category.
* The primary category is determined by the status content, not enclosed
* in brackets.
* The secondary category is determined by the status content enclosed in brackets.
* If not present, it is set to "*".
* The comparison is case-insensitive.
*
* @param {string} result
* @returns {object} An object with primary and secondary categories.
*/
const classifyResult = (result) => {
 if (isNil(result)) {

  return {
    primary: RESULT_DEFAULT_CLASS,
    secondary: RESULT_DEFAULT_CLASS,
  };
 }
 if (result.toLowerCase().includes("(")) {
  const indexOfOpenBracket = result.indexOf("(");
  const indexOfCloseBracket = result.indexOf(")");
  return {
    primary: result.substring(0, indexOfOpenBracket).toLowerCase(),
    secondary: result
      .substring(indexOfOpenBracket + 1, indexOfCloseBracket)
      .toLowerCase(),
  };
 }

 return {
  primary: result.toLowerCase(),
  secondary: RESULT_DEFAULT_CLASS,
 };
};

const prepareRows = (
 rows,
 toolIdx,
 categoryAccessor,
 statusAccessor,
 formatter,
) => {
 return rows.map((row) => {
  const cat = categoryAccessor(toolIdx, row);
  const stat = statusAccessor(toolIdx, row);
  const mappedCat = cat.replace(/-/g, "_");
  const mappedRes = classifyResult(stat);

  return {
    categoryType: mappedCat,
    resultType: mappedRes,
    row: row.results[toolIdx].values,
  };
 });
};
```

The following is a prospective implementation of the generation of the statistics row headers:

```
/*
* Generate the static rows for the statistics table based on the given data.
*
* @param {object[]} data The data to generate the static rows for. It is an array
* of objects, where each object has the following structure:
* {
*  "categoryType": string,
*   "resultType": {
*     "primary": string,
*     "secondary": string
*   }
* }
*
* @returns {object} The generated static rows.
*/
const generateStaticRows = (data) => {
 const uniqueRowData = new Set();
 data.forEach((row) => {
   uniqueRowData.add(row);
 });

 const staticRows = {
   total: {
     title: "all results",
     description: "Total number of results",
   },
 };

 const distinctLevel1Values = new Set();
 const distinctLevel2Values = new Set();

 // Add the data for level 3
 uniqueRowData.forEach((row) => {
   distinctLevel1Values.add(row.categoryType);
   distinctLevel2Values.add({
     categoryType: row.categoryType,
     primary: row.resultType.primary,
   });

   staticRows[
     `${row.categoryType}_${row.resultType.primary}_${row.resultType.secondary}`
   ] = {
     title: `${row.categoryType} ${row.resultType.primary}
             ${row.resultType.secondary}`,
     indent: 3,
     description: `Results where the property is ${
       row.categoryType
     } and the result is ${row.resultType.primary} (${
       row.resultType.secondary === RESULT_DEFAULT_CLASS
         ? "default"
         : row.resultType.secondary
     })`,
   };
 });

 // Add the data for level 2
 distinctLevel2Values.forEach((row) => {
   staticRows[`${row.categoryType}_${row.primary}`] = {
     title: `${row.categoryType} ${row.primary}`,
     indent: 2,
     description: `Results where the property is ${row.categoryType} and the
                   result is ${row.primary}`,
   };
  });

  // Add the data for level 1
  distinctLevel1Values.forEach((row) => {
    staticRows[row] = {
      title: row,
      indent: 1,
      description: `Results where the property is ${row}`,
    };
  });
  return staticRows;
 };
```

This approach has a minor issue: the descriptions generated might not always be the most descriptive and may feel mechanical. Still, we can easily overcome this by manually writing them for some of the most common use cases/categories and only generating the same dynamically when encountering a new item.

Changing the run set names to be more user-friendly

This is a relatively minor issue that I would take up next, as it should be a relatively quick bug fix.

Expected Time of Completion

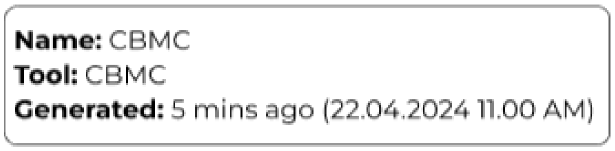
0.5 weeks

Approach

There is currently an existing utility function called *getRunsetName*, which always adds the timestamp of the runset to the name. Instead of using the same, I would like to change the way the name is displayed in the UI:

- If the names of all the runsets are unique, then no timestamp is added to their name
- If the name is not unique, then instead of appending the timestamp directly to the name, we will format the timestamp in either of the following formats before adding it to the end of the name:
    - Time since now format: This, in my opinion, is the most human-readable format to show the timestamps, where you convey to the user how much time has passed since the runset (like "4 minutes ago", "1 day ago," etc.)
    - A standard "DD:MM:YY HH:MM:SS" format instead of the long and often confusing ISO string format

In addition, I would like to add a tooltip to the name of the runset so that whenever a user hovers over the same, we can show some data about the runset to the user in a legible and easy-to-read manner. The following is a small possible UI of the tooltip:


Name: CBMC
Tool: CBMC
Generated: 5 mins ago (22.04.2024 11.00 AM)

Syntax highlighting for overlay panes

The overall plan to achieve this would include adding the Prism React Renderer package to the application to support the same, which is based on the (arguably) most popular syntax highlighting library today, prism.js. (See appendix)

Expected Time of Completion
1 week

Approach
We need to change the TaskDefinitionViewer component and surround it with the wrapper provided by the Prism React Renderer to display the appropriate syntax highlighting for the file content. We can additionally provide the user options to toggle some settings of the viewer, like:

- The theme of syntax highlighting
- The presence of the line number of the file content
- Editor styling preferences include line height, font, etc.

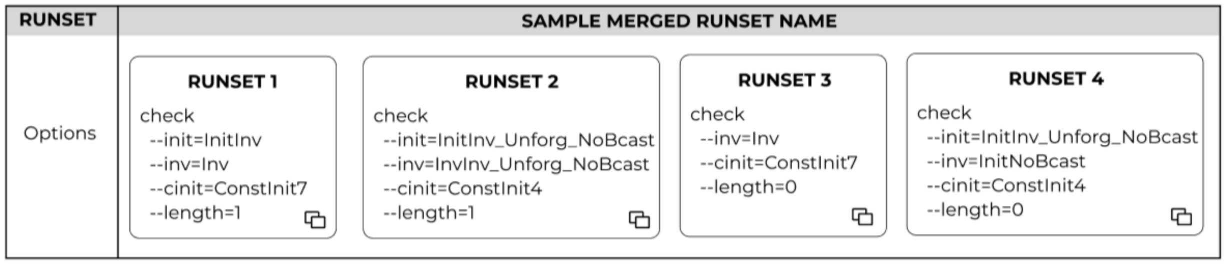
All of these require just creating another modal that allows the user to select all these options and preferences when opened. We can even sync these preferences for the user between sessions using local browser storage. Providing support for all these features would require passing props to Prism React Renderer or using the various plugins that come with the Prism ecosystem.

More intuitive UI for the Display of the Command Line Options in the case of Merged Runsets

This is one of the final enhancements I would like to make to the application in the last phase of the contribution. I plan to iterate on a few designs on Figma with the mentors to determine which would fit this use case the best and then spend the rest of the time implementing it.

Expected Time of Completion
1 week

Approach
The UI needs to have a way to allow readers to determine quickly which run set has which options. Moreover, semantic and accessible HTML tags should be used to render all the different run sets as a list. I plan to create a separate component to handle this rendering of merged run sets. The first iteration of the design of the same looks something like the following:



RUNSET	SAMPLE MERGED RUNSET NAME			
	RUNSET 1	**RUNSET 2**	**RUNSET 3**	**RUNSET 4**
Options	check --init=InitInv --inv=Inv --cinit=ConstInit7 --length=1	check --init=InitInv_Unforg_NoBcast --inv=InvInv_Unforg_NoBcast --cinit=ConstInit4 --length=1	check --inv=Inv --cinit=ConstInit7 --length=0	check --init=InitInv_Unforg_NoBcast --inv=InitNoBcast --cinit=ConstInit4 --length=0

We can create a card for each runset, which will only be responsible for showing the command line options that are related to it. The card would also have a Copy Icon on the bottom, allowing users to copy the command line instructions to the clipboard for easy access. We will display all the runsets in a horizontal row so that it is easier for the user to compare the differences in the command line options of the merged runsets.

Add Links to Raw Results to HTML Tables

Aims to address issue #233 on the GitHub issue tracker for BenchExec.

Expected Time of Completion

1-week

Approach

As there can be several XML files per table, adding links just to the result XML into the HTML table headers might not be the most feasible solution. Instead, I would like to create another overlay pane named **Raw Data**. This pane would be reachable via the header of the table. It would contain the links to all result XML files, logfiles ZIPs, CSV version of the table (if generated), and all information on how the table was generated (table XML if it was used, command-line flags). This would provide a good point for the user to view all the associated metadata with the table and information used to generate it. Thus, it would encourage reproducibility for users to publish their raw results alongside their tables.

Expected Results

I plan to complete all the tasks listed above, as well as close the linked issues to them. I also aim to ensure that all the new changes that have been added have 100% test coverage and clear JSDoc documentation. In addition to the unit tests, I plan to perform extensive manual regression testing to ensure that the existing functionality does not break due to refactoring during the upgrade of the React Router or by adding support for various status types to the table summary.

Timeline

I aim to make weekly to biweekly PRs for the project so that there is enough time for the mentors to review all the changes and suggest improvements before the next sprint cycle (The only exception to this might be the case where I need to add all of the functionality in one chunk otherwise it may break the application). This would also ensure we incrementally develop the component continuously and do not block progress due to unstable requirements and changes for unpredictable reasons.

To keep track of the project's progress and convey the changes to the developers, I would like to maintain a blog that would briefly touch upon the changes made and link the relevant PRs for implementation. It can also be a good place to reference the technical decisions made during the implementation. I plan to host the same on my website, much like last year's GSoC ‘23 completion blog. I would make a blog entry for every sub-project completion and a final blog at the end of the GSoC term that summarizes all the changes.

I plan to stay in close touch with my mentors via any comfortable means of communication (Google Meets, Slack, or WhatsApp). I would regularly update the mentors about my progress and any possible blockers via text (an update approximately every 3-5 days). I would sync with them biweekly via a meeting to review the progress and make changes to the timeline if necessary.

Period	Sub Project	Tasks
Week 1 - 2 [May 29 - June 10]	*Upgrading React Router*	- Start migrating to React Router 6 by refactoring the Switch component to the Route component. - Take an overview of how the parameters are handled across the codebase, and based on it, create the custom navigation hook with the supporting options
Week 3 [June 11 - June 18]		- Refactor the code base to make use of the custom hook everywhere and manually test all the existing features - Remove the existing unused utilities, and add tests for the custom hook
Week 4 - 5 [June 19 - July 1]	*Combining the Two Summary Tables into a Single Tabular Component*	- Create a new component for display of the merged UI and add the same to a new page for testing - Complete the component to display all of the basic information
Week 6 - 7 [July 2 - July 15]		- Work on implementing the Settings panel that would host all the settings for the display of the table - Integrate functionalities such as hiding columns, hiding information rows, and striped rows
Week 8 [July 16 - July 23]		- Work on improving the styling of the table - Ensure that the table works well on a variety of screen sizes - Add freezing functionality to the specified rows
Proposed Midterm Evaluation (20 July - 24 July)		

Week 9 - 10 [July 24 - August 12]	*Adding Support for Custom Error Types and the New Filtering Logic*	- Understand the structure and the parsing of the results in the *stats.js* file - Begin the upgradation of the same to parse the rows into the various categories as described
Week 11 - 13 [August 13 - Sept 3]		- Unit test the scripts developed - Make changes to the Summary UI to integrate the newly modified statistics - Add documentation related to the component and the logic for the classification
Week 14 - 15 [Sept 4 - Sept 17]		- Change the filter table UI to make use of the React Checkbox Tree library to display the hierarchical classification - Implement the handlers to perform the filtering
Week 16 - 18 [Sept 18 - Oct 8]	*Buffer Slot & Extended Goals*	- Complete any backlog tasks from the previous weeks - Work on any of the extended goals mentioned above in the remaining of the time
Proposed Final Evaluation (6 Oct - 10 Oct)		
Celebrate 🎉		

About Me

I am a full-stack web developer with experience with multiple tech stacks on both the front and back end, and I have a particular interest in testing and automation. I have been fortunate enough to contribute to numerous OpenSource organizations, which have helped me diversify my tech stack!

Beginner	**Intermediate**	**Advanced**
Django	GraphQL, Rust, React.js, Next.js, MongoDB, Python	Javascript, Typescript, Vue.js, Nuxt, Node, PostgreSQL, GoLang, Express

I am a responsible individual and a great team player who believes in the utmost importance of active communication. I was also a **GSoC '23 Mentee under the Palisadoes Foundation**. I worked for **22 weeks** on the Event and Volunteer Management project, where I learned in depth about React, GraphQL, and testing! I am still an active community member there and have **reviewed over 65+ PRs** for them ever since my GSoC ended. I have also worked with other popular OpenSource organizations in my free time, including Learning Equality, Appwrite, Plone, Harbor, Jaeger, and Vitess.

I have also **contributed to the BenchExec project previously**. Here are the links to my contributions:

- Introduction of "decodeFilter" function: #1005
- Refactor "setHash" and related functions to improve readability: #1006

- Adding a suggestion to replicate the GitLab pipeline of GitHub and adding the support for the same: #1007, #1009

I have ample experience with web development, testing, and building microservices; thus, I am confident that I should be able to navigate the requirements of this project. I am also a good communicator and a team player who is always open to feedback.

Commitments
I have no other commitments for the upcoming summer and the GSOC timeline. After completing my seventh semester, I may have institute exams, but the same would be over in under ten days. I am well versed in academics and can ensure that the same would help my dedication and work schedule.

Plans after GSoC
I love OpenSource, and working with the BenchExec project would be a dream come true. I would love to continue to be an active community member here as a code contributor and a guide who can help other Open Source enthusiasts stumbling upon this project and the SOSY ecosystem.

Planning for the Development Period:
I would like to work with the mentors to decide weekly goals to align myself and work in short sprints to build the required functionality. The timeline for the project, as mentioned earlier, would help me guide the overall development.

I am a good communicator, so I want to stay in close sync with the mentors via texts, Slack, or meetings to clear all the blocking factors quickly and focus more on development. I am a night owl, so I plan to devote 150-180 minutes (2.5 - 3 hours) every night, seven days a week, to the project for 18 weeks, totaling up to 360 hours of work. I would happily put in extra hours if the project demands the same. This would accommodate all the buffer periods, personal leaves, and the time required for testing and documentation.

Appendix

Why prism.js?
Since no central authority can decide the “most popular” library for syntax highlighting, we often need to go with the community favorites and libraries with the most support. A quick Google search about the same would lead to a seemingly endless number of blogs, whether everybody shares their opinions, all with different recommendations. But in all of them, it is almost certain that the names of two libraries would pop up: highlight.js and prism.js.

I advocate for prism.js as many popular websites are using it today:
- CSS Tricks
- Smashing Magazine

- Sitepoint
- Stripe
- Drupal

It offers some great features as well:

- **Good Practices:** Prism promotes using the correct HTML elements for marking up code, such as <code> for inline code and <pre> for code blocks.
- **Web Workers Support:** It supports parallelism with Web Workers, although it is turned off by default.
- **Easy to Extend:** It has a plugin architecture, making it easy to extend without modifying the core code. Multiple hooks are available throughout the source.
- **No Specific Markup Required:** Prism does not force using any specific markup or class name.
- **Comprehensive Browser Support:** Major browsers, including Edge, IE11, Firefox, Chrome, Safari, Opera, and most mobile browsers, support Prism.

highlight.js is another excellent option (arguably more popular if we consider GitHub stars on the repository as a measure of popularity). Still, I lean towards Prism due to my previous experience with it and the fact I have never had a problem with it before.