iTrace-Toolkit: A Pipeline for Analyzing Eye-Tracking Data of Software Engineering Studies

Joshua Behler
Department of Computer Science
Kent State University
Kent, Ohio, USA
jbehler1@kent.edu

Praxis Weston
Department of Computer Science
Kent State University
Kent, Ohio, USA
gweston2@kent.edu

Drew T. Guarnera
Department of Mathematical and Computer Sciences
College of Wooster
Wooster, Ohio, USA
dguarnera@wooster.edu

Bonita Sharif
School of Computing
University of Nebraska-Lincoln
Lincoln, Nebraska, USA
bsharif@unl.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio, USA
jmaletic@kent.edu

***Abstract*—iTrace is community eye-tracking infrastructure that enables conducting eye-tracking studies within an Integrated Development Environment (IDE). It consists of a set of tools for gathering eye-tracking data on large real software projects within an IDE during studies on source code. Once the raw eye-tracking data is collected, processing is necessary before it can be used for analysis. Rather than provide the raw data for researchers to analyze and write their own customize scripts, we introduce iTrace-Toolkit - a suite of tools that assists with combining different data files generated from iTrace and its IDE plugins (namely Visual Studio, Atom, and Eclipse). iTrace-Toolkit also provides the crucial mapping of the valid raw eye-tracking data to source code tokens and finally generates fixations (an important metric in eye-tracking for comprehension) using three commonly used algorithms based on distance and velocity of eye movements. iTrace-Toolkit keeps track of all participant data and tasks during a given study and produces a complete lightweight database of the raw, mapped, and fixation data that is standardized and ready to be used by statistical tools. A simple GUI interface is provided for quick access to filter the data after an eye-tracking study. iTrace-Toolkit also allows for the export of the data or subset of the data to text formats for further statistical processing.**
YouTube Video:
https://www.youtube.com/watch?v=9j2OsOANh8w

I. Introduction

The iTrace eye-tracking infrastructure [1], [2] has been used by software engineering researchers to aid with their studies on real code spaces. Previously, eye-tracking studies on software development would require the participant to view a static image of code. While this works, it can feel unnatural to a participant when they are more familiar with code appearing in a common IDE like Visual Studio or Eclipse along with the ability to scroll and look back and forth at multiple files. iTrace alleviates this problem by providing iTrace-Core and the iTrace IDE Plugins. By using iTrace-Core along with an IDE Plugin of choice, studies can be performed with participants looking at a familiar workspace while also allowing dynamic scrolling and changing the files read. This process also helps researchers conduct their research faster, as code no longer needs to fit on screen dimensions nor needs to be converted to images for viewing.

Both iTrace-Core and the iTrace Plugins output large XML files containing a list of the information gathered during the eye-tracking session—iTrace-Core gathering the raw gaze data, and the Plugin gathering IDE contextual information. An issue that researchers face with this data is in analyzing the output for what they want. A simple five-minute session can result in 35,000+ gaze and contextual data points saved in roughly 30 MB of XML files. Analysis of this data is not only difficult, but also non standardized, meaning that each individual researcher will have to implement their own software to go through the data to do any research tasks such as mapping gazes to tokens, calculating fixations and saccades, and filtering data points based on the research subject.

To help researchers analyze their data, we introduce iTrace-Toolkit, an application focused on combining, mapping, analyzing, and filtering the output from iTrace-Core and iTrace Plugins. iTrace-Toolkit offers the following features to software engineering researchers:

- **Source Code Token and Context Mapping:** The iTrace Plugin data contains file line and column information that is gathered from the IDE but, beyond that, does not provide any additional information. iTrace-Toolkit captures the entire word or symbol located at the line and column and provides the relevant syntactical context of the token. This information is crucial for studies on program comprehension for example.
- **Fixation Generation:** Eye trackers and the iTrace Plugins are designed to produce and collect eye raw gazes. In order to make sense of the data, fixations need to be computed. A fixation happens when the eye stabilizes on a certain part of the stimuli (e.g., source code) for a given duration i.e., implying the user is reading/comprehending some aspect of the stimuli. Fixations are a set of raw gazes in a certain area in space and time. Using iTrace-Toolkit and various pre-defined fixation algorithms [3], researchers can choose to generate and store fixation information.
- **Fixation Filtering:** A filtering option is provided by iTrace-Toolkit to assist researchers quickly make sense of the data. This feature is very useful to do quick checks on data collected especially if a researcher is expected

to see differences in the code between the different treatments in the study. It is also useful to filter out fixations based on the type of research question being asked. We use this feature in our own studies to quickly make sense of the data right after a study.

This paper is organized as follows. In the next section, we give an overview of the complete iTrace Infrastructure, of which iTrace-Toolkit is part of. Next, the implementation of iTrace-Toolkit is presented in Section III. The basic functionality of iTrace-Toolkit is given in Section IV along with usage scenarios in Section V. The final section, Section VI, provides conclusions and future directions.

II. THE iTRACE INFRASTRUCTURE

iTrace-Toolkit is designed to be used within the greater iTrace Infrastructure [1], [2], as detailed in Figure 1, to ease a researcher's burden of analyzing their data. A researcher sets up and conducts their study using iTrace-Core and one of the various IDE Plugins and collects all the outputted Core and Plugin data files from each of their participants doing a set of tasks. Optional tools such as the iTrace-Core built-in Deja Vu tool can be used to get accurate data if high-speed eye trackers (>300Hz) are used [2]. After gathering all the files from the researcher's sessions, iTrace-Toolkit is used to create a database, and all the Core and Plugin files are brought into the database. It is best practice to include all the recording sessions of a study that share a target code space – i.e., if 20 participants looked at program A and 30 participants looked at program B, two databases should be made—one for the A sessions and one for the B sessions. iTrace-Toolkit allows for previously made databases to be imported as well, giving the researcher full flexibility in how they want to keep track of their data.

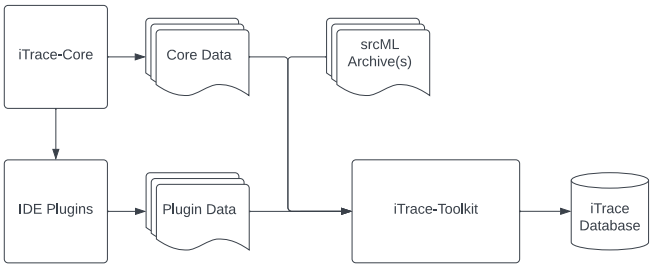

Figure 1: An overview of the iTrace Infrastructure, and how the iTrace-Toolkit fits within it.

Some tools offer a similar level of analysis but are limited in their scope or ability to analyze. For example, our previous work on the gazel tool [4] can help analyze data generated with the iTrace-Atom Plugin, but is unable to work with the other iTrace Plugins.

III. iTRACE-TOOLKIT IMPLEMENTATION

iTrace-Toolkit is implemented using a variety of technologies to store and transform the large amounts of data produced during an eye-tracking study. Note that a study can easily collect close to a few GBs of data with 30 participants working on 3 tasks each for instance. The amount of data generated is also highly dependent on the eye tracker speed (how many samples are generated per second-iTrace supports all speeds), amount of code viewed, and time to complete the task. The current implementation is in C++ with an SQLite database. Please note that iTrace-Toolkit only supports Windows currently. A publicly available tool with usage instructions and example data is available at https://github.com/iTrace-Dev/iTrace-Toolkit/releases/tag/alpha-0.2.1

A. Implementation of Technologies

iTrace-Toolkit has gone through numerous design changes during its development. iTrace-Toolkit originally started out as a Tkinter program written in Python. It quickly became apparent that more efficiency was necessary to address the size of the data. We settled on using the Qt platform with C++ to provide both faster speeds along with more modern GUI tools. One downside of using Qt is that the Qt library for interacting with an SQLite database is rather slow. To fix this, we opted to use the direct C library for SQLite. These changes, along with tactful indexing of the database, gave us an almost 11x speed increase. iTrace-Toolkit went from taking five and a half hours to map the tokens of 13 tasks down to just 30 minutes.

B. iTrace Database

Before any gaze or context data can be analyzed or even imported, an iTrace Database needs to be created. iTrace-Toolkit makes use of SQLite to store the imported data. All analysis performed is also done on this database. iTrace-Toolkit allows a new, empty database to be created, or for a previously made database to be opened and used again.

Specific information on the tables and their columns can be found on iTrace-Toolkit's GitHub wiki page[1]. The three most important tables are `gaze`, `ide_context`, and `fixation`. The `gaze` table stores the raw gaze data produced from iTrace-Core. The `ide_context` data saves the information from the Plugin file as well as token mapping information. The `fixation` table is where fixations are stored after calculation.

IV. iTRACE-TOOLKIT PIPELINE

The design of iTrace-Toolkit is divided into separate tasks which compute and transform raw eye tracking data into more abstract and useful concepts.

A. Importing Core and Plugin Data

After an iTrace database is created, iTrace-Core and Plugin data can be imported. Core and Plugin data must be imported in pairs, one for each session. A folder containing multiple pairs can be selected, allowing for bulk importing. After the sessions are imported, a list of each session's id and name is listed, and a checkbox is shown next to each session. The checkboxes allow a user to pick which sessions to use for mapping or for fixation generation—any unchecked session will be ignored.

B. Token Mapping

One of iTrace-Toolkit's primary features is the ability to map the line and column information to a token in the source code.

[1] https://github.com/iTrace-Dev/iTrace-Toolkit/wiki/Entity-Relationship-Diagram-for-Post-Processing-Database

This is an extremely crucial part of analyzing eye tracking data on source code. Researchers are interested in what words or symbols a participant looks at during a session. iTrace-Toolkit allows a researcher to calculate these tokens by providing the source code that is examined during the session. iTrace-Toolkit leverages srcML[2] [5], [6] to accomplish this task. srcML is an infrastructure to support the analysis, exploration, and manipulation of source code. It produces an XML representation of source code that provides abstract syntactic information. It also provides both context and positional information of the source code, which is vital for our token mapping. srcML currently supports C, C++, C#, and Java.

iTrace-Toolkit locates the corresponding srcML representation for the file which is being examined. It will then find the srcML tag located at the correct line/column. Because line/column data represents only one character, iTrace-Toolkit will also find the whole token by first figuring out which type of token is being examined at–either whitespace, a word, or a symbol such as an operator. It will then march left and right until it finds a character that does not fit within that type. This token will be recorded inside the `ide_context` row that provides the line/column data. It is important to note that if the token happens to be whitespace, the whitespace itself will not be recorded, and it will instead be saved as `WHITESPACE` in the database.

```
1  #include <iostream>
2
3  int main() {
4      return 0;
5  }
```

Figure 2: A small example of C++ code with a raw gaze point (highlighted)

Along with the full token, iTrace-Toolkit records contextual data about what the user is viewing and stores it. Consider the C++ code in Figure 2. The yellow-highlighted character is at line, column position (3, 6), and represents a point of data in the `ide_context` table. We can see that the user is looking at the function name, `main`, which is recorded in the token column. Contextually we know that `main` is the name of a function inside the file (referred to as a `unit` by srcML), and by using srcML we can record this context in the table as well. For this example, the user is looking at the name of a function at the top level of a file. We store this data in two columns in the database, `source_token_xpath` and `source_token_syntactic_context`. The syntactic context column stores an arrowed list of srcML tags that describes where the text is located contextually. The syntactic context of this example would be `unit->function->name`. The XPath column stores an XPath query that leads to the exact tag in the srcML. The XPath for this example is (assuming the file name is `file.cpp`):

```
//src:unit[@filename='file.cpp']/
src:function[@pos:start='3:1' and
@pos:end='5:1']/src:name[@pos:start='3:5' and
@pos:end='3:8']
```

Because of the additional positional information that is stored, the XPath is always unique to the specific token that is being examined, while a syntactic context can refer to multiple entries within the database.

C. Fixation Generation

iTrace-Toolkit's other primary feature is the ability to group together gazes to form fixations. Eye movements are characterized by saccades, rapid darts between objects, and fixations, focuses on objects. Because an eye tracker only gets so many snapshots of an eye (e.g., 60, 120, … data points per second), it is important to know if a gaze is part of a saccade or a fixation. iTrace-Toolkit implements three different algorithms to calculate fixations, Basic, I-VT, and I-DT [3], [7]. After generating fixations, the `fixation`, `fixation_gaze`, and `fixation_run` tables in the database are populated with data related to the fixation generation run. Sessions can have multiple fixation generation runs performed on them, allowing comparison between algorithms and their various settings.

If token mapping is performed before running the fixation run, fixations can be mapped with the `ide_context` list and grab the token information that matches with the gazes. If no valid information is found the fixation will instead contain blank data indicating eye movements on something other than source code. While fixations can be generated without token mapping, any resulting fixations will lack token information.

iTrace-Toolkit is designed to allow for modular fixation algorithm implementation. If future works or individual researchers devise new fixation detection algorithms, iTrace-Toolkit eases the process of implementing and integrating new algorithms. We provide the three most popular fixation detection algorithms but welcome the community to add more as needed as there might be cases where a study might require a different method of generating fixations.

D. Fixation Filtering

After fixations are generated, iTrace-Toolkit offers tools to help researchers gather the fixations they are studying. Fixations can be filtered by various factors, such as pupil diameter, file name, token type, file line and column, and fixation duration to name a few. These filters can be imported and exported for more fine-tuned filtering and for use outside of Trace-Toolkit respectively. Figure 3 showcases the various options available for fixation filtering. The filtered fixations can be saved in a .db3, .tsv, .json, or .xml file format, and can undergo whatever further statistical analysis the researcher needs in a statistical package of their choosing.

V. Usage Scenarios

Using iTrace-Toolkit involves first running an eye-tracking study using iTrace Core and one of the IDE plugins. Once the researcher has finalized the design of the study and tasks, they typically start data collection. During data collection, human participants (developers) are presented with the task(s) and source code project. iTrace allows for large code bases to be explored in an IDE within the context of a software engineering task. Participants can freely switch between multiple files and

[2] See www.srcML.org

scroll within a file. After a participant completes the study, the resulting data is used as input into the iTrace-Toolkit. The data includes the raw eye gaze data and the corresponding line, column, and file information.

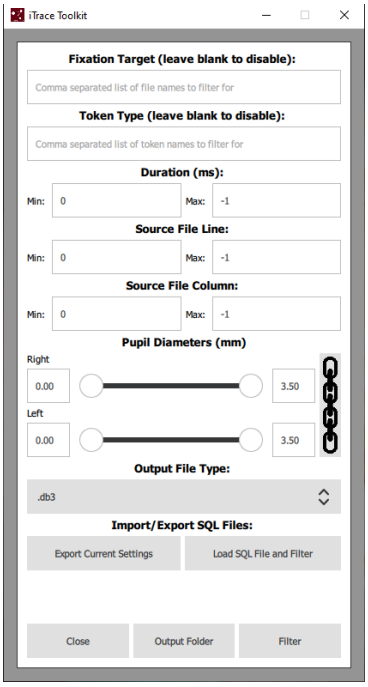

Figure 3: Fixation Filtering Options in iTrace-Toolkit.

Processing and filtering eye tracking data takes a substantial amount of time. iTrace-Toolkit alleviates this burden on the researcher and allows them to focus on their research rather than the tool itself. iTrace-Core and the iTrace Plugins can be used for conducting studies involving many different software tasks such as code summarization, bug localization, and code review. iTrace-Toolkit can then be used to process, map, analyze and filter the rich data set generated from these tasks, which is the most time-consuming part of studies.

After the data is imported into the database, line and column coordinates are mapped to tokens. Token mapping can be the longest step of using iTrace-Toolkit, and long studies with larger code bases took about 30 minutes in our latest test. After the tokens are mapped, fixations are generated, a researcher can either manually analyze the fixations in the database or use the built-in filtering tools to select specific fixations within specified criteria. After the researcher generates fixations and sets the filtering parameters how they want, they can choose between various output formats depending on their needs. Filters can be exported and imported, allowing for reuse later if multiple databases are needed. The output file will contain the fixations that fit the criteria provided in the filtering window. Typically, at this point, the researcher will export all the data they need out of iTrace-Toolkit for further statistical analysis based on their specific research questions. After this, the researcher is free to analyze the exported information in any way they need for their research.

VI. Limitations and Future Work

Currently, iTrace-Toolkit only supports the iTrace IDE Plugins (Visual Studio, Atom, and Eclipse). Additionally, iTrace-Toolkit is limited in token mapping by srcML's support of programming languages. Currently, C/C++, Java, and C# are supported by srcML. However, support for other languages is planned by the srcML project. Studies that involve languages other than those supported can still use iTrace-Toolkit to generate fixations, but any token or syntactic information will need to be analyzed separately.

Plugin data generated by iTrace-Chrome, the iTrace Plugin for the Google Chrome Web Browser, cannot be analyzed by iTrace-Toolkit due to iTrace-Chrome's data being ad-hoc based on the webpage being viewed. The iTrace-Chrome data needs to be analyzed with the support of ad-hoc specialized scripts. As part of future work, iTrace-Toolkit will be expanded to allow for iTrace-Chrome data to be imported for fixation analysis, as well as provide a custom system to analyze the tokens on web pages.

iTrace-Toolkit also currently does not calculate any saccade [3] or microsaccade [8] information, which is useful for determining cognitive load. Additionally, iTrace-Toolkit can be improved further by implementing more fixation detection algorithms made specifically for source code, as it has been shown that developers read source code very differently from natural language. A command line interface for iTrace-Toolkit is also planned as part of our future work, along with versions for both Mac and Linux based Distros.

References

[1] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, "iTrace: eye tracking infrastructure for development environments," in *10th ACM Symposium on Eye tracking Research and Applications*, Warsaw, Poland, Jun. 2018, p. 3. doi: 10.1145/3204493.3208343.

[2] V. Zyrianov *et al.*, "Deja Vu: semantics-aware recording and replay of high-speed eye tracking and interaction data to support cognitive studies of software engineering tasks—methodology and analyses," *Empir Software Eng*, vol. 27, no. 7, p. 168, Dec. 2022, doi: 10.1007/s10664-022-10209-3.

[3] D. D. Salvucci and J. H. Goldberg, "Identifying Fixations and Saccades in Eye-tracking Protocols," in *2000 Symposium on Eye Tracking Research & Applications*, Palm Beach Gardens, Florida, USA, Nov. 2000, pp. 71–78. doi: 10.1145/355017.355028.

[4] S. Fakhoury *et al.*, "gazel: Supporting Source Code Edits in Eye-Tracking Studies," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Madrid, ES, May 2021, pp. 69–72. doi: 10.1109/ICSE-Companion52605.2021.00038.

[5] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration," in *2013 IEEE International Conference on Software Maintenance*, Eindhoven, Netherlands, Sep. 2013, pp. 516–519. doi: 10.1109/ICSM.2013.85.

[6] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, Williamsburg, Virginia, USA, Sep. 2011, pp. 173–184. doi: 10.1109/SCAM.2011.19.

[7] P. Olsson, "Real-time and Offline Filters for Eye Tracking," Masters Thesis, KTH Electrical Engineering, Stockholm, Sweden, 2007. Accessed: Jun. 21, 2019. [Online]. Available: https://pdfs.semanticscholar.org/4167/7844556582adc68a5a14dbb1cea0b28d9016.pdf

[8] R. Engbert and R. Kliegl, "Microsaccades Keep the Eyes' Balance During Fixation," *Psychol Sci*, vol. 15, no. 6, pp. 431–431, Jun. 2004, doi: 10.1111/j.0956-7976.2004.00697.x.