RacketLogger: Logging and Visualising Changes in DrRacket

Turgut Reis Kursun
turgut.reis.kursun@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Jens Van der Plas
jens.van.der.plas@vub.be
Software Languages Lab, Vrije Universiteit Brussel
Brussels, Belgium

Quentin Stiévenart
quentin.stievenart@vub.be
Software Languages Lab, Vrije Universiteit Brussel
Brussels, Belgium

Coen De Roover
coen.de.roover@vub.be
Software Languages Lab, Vrije Universiteit Brussel
Brussels, Belgium

ABSTRACT

Developers frequently make code changes while programming, such as deleting a line of code and renaming or introducing a variable. These changes can be detected and logged, for example by the IDE used by the developer. Logging changes is possible at two levels: at the textual level or at the level of the abstract syntax tree (AST) of the program. The logged changes, in both forms, are useful because they can be used to build new software engineering tools, such as static code analysers.

Plugins that log changes have already been developed for some IDEs. However, so far, no change-logging plugin has been developed for the DrRacket IDE, which supports the development of programs written in Scheme-like languages such as R5RS Scheme and Racket. To fill this gap, we have developed RacketLogger, a change-logging plugin for DrRacket. RacketLogger logs changes both at the textual level and at the AST level. To determine changes at the level of the AST, we have adapted Negara et al.'s algorithm to support Scheme syntax. We have evaluated our plugin by creating a visualisation for the logged changes to measure how well RacketLogger can be used as a building block, and conducted a small-scale user study to measure its usability.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**.

KEYWORDS

Racket, change logging, IDEs

ACM Reference Format:
Turgut Reis Kursun, Jens Van der Plas, Quentin Stiévenart, and Coen De Roover. 2022. RacketLogger: Logging and Visualising Changes in DrRacket. In *Proceedings of the 15th European Lisp Symposium (ELS'21).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.6326894

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'21, March 21–22, 2022, Genova, Italy
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.6326894

1 INTRODUCTION

During development, developers make multiple changes to their program. Many types of changes can occur: inserting new code, deleting a variable, applying a refactoring, and so on. To support development, usually, changes are tracked with version control systems, such as git. The changes logged by version control systems are unsatisfactory for certain applications, as they only provide snapshots of committed code. This raises the need for a more immediate tracking of changes, allowing changes *between* commits to also be stored. As a solution, a *change logger* can be used: a change logger logs all changes that happen during development. Having a lower-level view on the changes enables new applications such as programming pattern detection [1, 5, 9, 10, 14], and tools for collaborative software development [3].

Different granularities and representations can be used for the logged changes. A change logger is called fine-grained if it logs changes at a detailed level, so that almost every interaction is logged. Such fine-grained change loggers can reconstruct every intermediary state of the source code using the logged data [6, 14]. On the other hand, change loggers that only log certain interactions are called coarse-grained.

Changes can be logged at two different levels: at the textual level, and at the level of the abstract syntax tree (AST). Logging changes at the textual level means a change denotes how the *text* of the program has changed. Such a logging mechanism may for example track every character insertion. Logging changes at the AST level means that changes are represented as operations on the nodes of the AST (e.g., inserting, deleting, or updating a node) that connect the AST at a given state of the program to the AST of the subsequent state of the program. Changes logged at the AST level are a rich source of information. They represent which subtrees of the AST have been subject to change, information that can for example be used by incremental program analyses [11, Ch. 7],[12, 13]. Of course, changes at the textual level and changes at the AST level are related, as one often needs the textual changes in order to compute the AST changes. However, AST changes can be obtained only when an AST can be constructed, that is, at points during the development of the program where the program can correctly be parsed.

Multiple change-logging plugins have already been developed. For example, efforts have been made by the research community to provide change loggers for IDEs such as IntelliJ [1] and Eclipse [14]. However, to the best of our knowledge, there does not yet exist a change-logging plugin for the DrRacket IDE.

In this paper, we present RacketLogger, the first change-logging plugin for the DrRacket IDE. RacketLogger —developed as part of a Bachelor Thesis [8]— is implemented in the Racket language and can log changes for any language that uses s-expression syntax –in the remainder of this paper, we will focus on the Scheme language. RacketLogger is a fine-grained change-logging plugin which logs changes textually, representing the actual edits made to the source code, and uses these changes to also infer changes at the AST level, which are persisted as well. To this end, we have adapted the change-inferencing algorithm of Negara et al. [6] to Scheme.

Section 2 discusses the two change representations adopted by RacketLogger in more detail. We then show how RacketLogger can be used as a building block to develop new tools by building a change visualiser (Section 3) which we evaluate by means of a user study (Section 4). Other potential usages of RacketLogger are discussed in Section 5. In Section 6, we review related work on change-logging. We conclude in Section 7.

2 REPRESENTING PROGRAM CHANGES

In this section, we discuss how changes to Scheme programs can be represented. RacketLogger stores the changes at a textual level but is also able to infer the AST node operations from these textual changes. First, in Section 2.1, changes at the textual level are described, before presenting the changes at the level of the AST, which we refer to as *AST node operations*, in Section 2.2.

2.1 Textual Changes

At the lowest level, RacketLogger logs all meaningful interactions with the DrRacket IDE. For that reason, and similar to other change-logging plugins [14], RacketLogger relies on a hierarchical representation of textual changes. The use of a hierarchy enables reasoning about the changes at different levels of abstraction. The hierarchy of changes used by RacketLogger is represented in Figure 1.

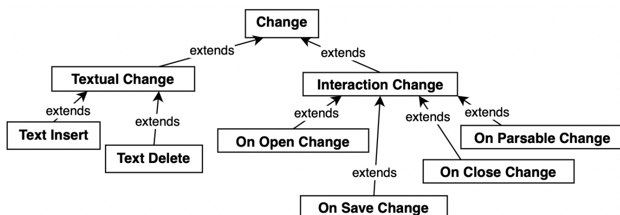

Figure 1: The change hierarchy used by RacketLogger

Changes are divided into two broad categories: *textual changes* and *interaction changes*. Textual changes impact the contents of the source code file, i.e., these are text inserts and text deletions. Interaction changes are non-textual changes, and represent interactions between the programmer and the IDE, such as opening, saving, and closing a file. They provide a context for the textual changes, such as in which file the textual changes were made. An *on parsable* change is logged whenever the code reaches a parsable state. This type of change is useful to trigger the inferencing of AST node operations, which relies on the code being in a parsable state.

2.1.1 Persisting changes. The textual changes logged by RacketLogger are persisted to a file, where each line corresponds to one change. An example log file is given in Listing 1. Each change is represented as a tagged list, where the first element, the tag, indicates the type of the represented change, and the remaining elements contain information about the change itself. For example, text insert changes contain the text that has been inserted and the offset at which the text was inserted. Storing changes as a tagged list is particularly useful in Racket, as this representation can easily be parsed and manipulated in Racket itself.

Listing 1: Example of persisted textual changes.

```
(text-insert ")" 39)
(on-parsable)
(on-close)
```

2.1.2 Merging changes. Logging changes at each keystroke is likely to result in large log files. When multiple characters are inserted or deleted at the same place in the source code, it is possible to merge these changes into a single change [6, 14]. Consider the addition of the character 'a' in a program represented by `(text-insert "a" 0)`, followed by the addition of the character 'n' `(text-insert "n" 1)`. These two changes can be merged into a single change, `(text-insert "an" 0)`.

RacketLogger automatically merges changes when possible, that is, when the following conditions are met:

- The textual changes must be of the same type, e.g., a text insertion change cannot be merged with a text deletion change.
- The changes have to be made consecutively in time, to avoid losing information about how changes were interleaved, e.g., two changes cannot be merged if another change has been made in between.
- Changes need to be made consecutively in space. This is the case if the second change starts at the offset where the first change has stopped. This means that if a character is added somewhere in the file, and then a character is added somewhere unrelated in the file, these two changes should not be merged.

Note that merging is performed transitively: a change that is the result of a previous merge can become part of a merge again, and hence, any number of changes can become merged into a single change as long as the above conditions are fulfilled.

As an example, consider Figure 2 which represents two textual changes, `(text-insert "ab" x)` and `(text-insert "c" x+2)`. Note that when we add the offset of the first change, `x`, to the length of its text, then the offset of the second change is obtained. As the second change starts at the offset where the first change has stopped, RacketLogger merges them. Similarly, two consecutive textual deletions can also be merged. In this case, the first change needs to be `(text-delete "c" x+2)` and the second change needs to be `(text-delete "ab" x)`.

2.2 AST Changes

Storing changes at the textual level may be too fine-grained or impractical for applications to work with. For this reason, RacketLogger is able to infer AST changes from the textual changes when

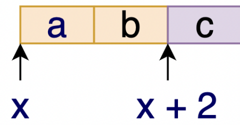

Figure 2: Two textual changes to be merged

the program is in a parsable state (so that an AST can be obtained). AST changes describe how the AST has changed from one parsable state to the next, and are computed as soon as the AST comes to a new parsable state. To see this, consider Figure 3, which exemplifies the derivation of AST changes from textual changes. Blue nodes denote node update operations, meaning that the contents of the node has been updated, and green nodes denote node insert operations, meaning that nodes are inserted. When no node operations are present in a subtree, this means that the subtree has remained unchanged between two parsable states. For example, in Figure 3, consider the subtree encircled in orange. Clearly, the AST changes are a more rich and interesting source of information — which can be used by program analyses run by the IDE for example — than the corresponding textual changes — which give no information on what nodes have been updated, inserted, deleted, or have remained unchanged.

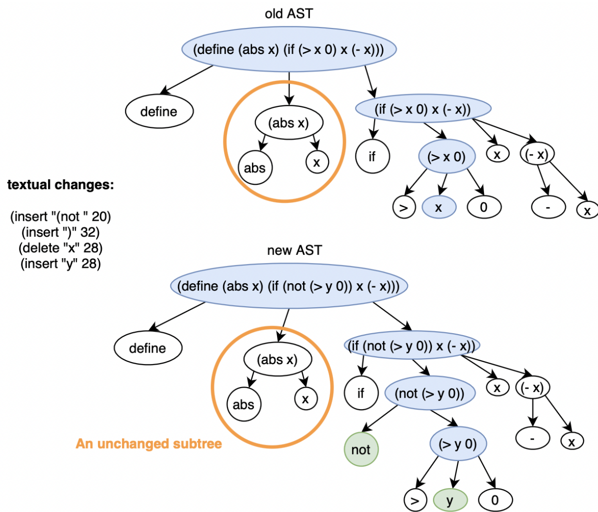

Figure 3: AST changes between two parsable states, showing node updates (blue) and node inserts (green). An unchanged subtree has been encircled in orange.

2.2.1 An Analogy for AST Change Inferencing Algorithms. AST changes are inferred from the last parsed AST before the changes, the current parsed AST, and the textual changes that connect them. To understand how the change-inferencing algorithm for AST operations works, we now first provide an intuitive analogy: the game of spot-the-difference, often played by children, and exemplified in Figure 4. The goal of the game is to find differences between two images. Analogously to the game, a change-inferencing algorithm finds the differences between two ASTs, in terms of node operations. More similarities emerge between inferencing algorithms and the game if we assume that the right image follows from the left image. This assumption allows the left image to play the role of an old AST, whereas the right image can play the role of a new AST. The deletion of an item in the left image corresponds to a node deletion operation in the old AST, and is indicated in black in the figure. Notice that the deleted item is present only in the first image, and similarly, a node deletion occurs only at a node of the old AST as it cannot be shown in the new AST. The insertion of an item into the second image corresponds to a node insertion operation into the new AST, and is indicated in blue. The inserted item is present only in the second image, and similarly, a node insertion occurs only at a node of the new AST. Finally, some elements that are present in both images are updated, which corresponds to node update operations. The updated elements are present in both figures, and similarly, updated nodes come in pairs: one in the old AST, one in the new AST.

Figure 4: Example of a spot-the-difference game. Blue circles mark updates to a part of the figure, green circles mark additions, and black circles mark deletions.

2.2.2 High-level overview of the Change-Inferencing Algorithm used by RacketLogger. To derive AST changes made to a Scheme program from textual changes, we have adapted the algorithm for the inferencing of AST node operations of Negara et al. [6]. This algorithm returns node operations (insertions, deletions and updates), given the old AST, new AST, and the corresponding textual changes that represent the changes to code when going from the old AST to the new AST. We first give an overview of the algorithm developed by Negara et al. Afterwards, we discuss how it was adapted for Scheme.

To generate node changes, the algorithm first establishes the root of the changed subtree, called the *common covering node*, which is present in both the old AST and the new AST. Finding it is of interest, since the rest of the algorithm can then operate on this subtree, saving computational efforts. Since such a common covering node represents the root of the changed subtree, it encloses all changes. In general, nodes are found by finding the traversal path from the root

of the AST to that node. Hence, finding the common covering node boils down to finding its path. The path is found in two steps. First, the algorithm looks for local covering nodes in both ASTs, these are the innermost nodes that enclose all textual changes. Second, the path to the common covering node is found by taking the common part of the paths to local covering nodes.

When the algorithm has found the common covering node, it starts matching descendants of the common covering node. The matching of nodes denotes the fact that the new AST node was already present in the old AST. Nodes are matched in two ways:

- The algorithm matches outliers, i.e., nodes that have not been affected by any changes. Hence, outliers remain unchanged and no node operations need to be generated.
- The algorithm matches yet unmatched nodes that have the same traversal path from their respective roots. For these nodes, update operations need to be generated.

Lastly, the algorithm generates node insert, node delete, and node update operations. For every unmatched descendant of the common covering node in the old AST, a delete operation is generated. For every unmatched descendant of the common covering node in the new AST, an insert operation is generated. For every pair of matched nodes whose content has changed, an update operation is generated. Notice how this generation of operations is consistent with our previous analogy.

2.2.3 Application to Scheme. Now that we have given an overview of the original algorithm developed by Negara et al. [6], we explain how we have adapted it to Scheme. First, we have noticed that the algorithm must generate more update operations. More precisely, an update operation must be generated for every pair of ancestors of the common covering node. The original algorithm generates operations only for nodes below the common covering node. Since the common covering node in the old AST and the new AST share their traversal paths, this means that they have an equal number of ancestors, who match with each other. These nodes enclose all changes since they enclose the common covering node, so they must have changed. Thus, an update operation must be generated for each pair of nodes on the path from the root of the AST to the common covering node.

Second, we have implemented low-level logic that enables the original algorithm to decide when a node is affected by a change. Note that when the algorithm matches outliers, it must be able to tell if a change affects a node. We have implemented this check for Scheme, and have noticed some peculiarities that relate to the syntax of the language. Consider Figure 5, where the code at the top corresponds to an identifier enclosed within parentheses and the code at the bottom corresponds to a simple identifier. Now, consider a textual insertion that occurs to the right of these nodes, indicated by the arrow. If a change starts at this offset, we see that the top node is untouched. However, the node representing the identifier node might be touched, as the name of the identifier may be made longer: when the added code does not start with a space or a parenthesis, the identifier is affected. Similar rules are required for insertions that occur at the beginning of the code represented by a node.

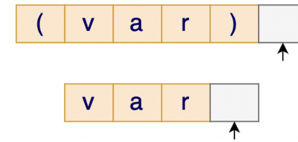

Figure 5: A bracketed S-exp, and an identifier S-exp. The arrow indicates the offset of a textual insertion

2.2.4 RacketLogger's Inferencing Algorithm in Pseudocode. Algorithm 1 shows the pseudocode for RacketLogger's AST node operations inferencing algorithm. The input to the algorithm is the old AST, the new AST, and the textual changes. These are the textual changes that took the code from the `oldAST` to the `newAST`. The output to the algorithm is a set of AST node operations, `ASTops`. These operations are update, insert, and delete operations. Recall that our algorithm is based on the state-of-the-art algorithm used by Negara et al. [6]. We have highlighted the parts of the algorithm which we adapted to Scheme. We will now discuss the algorithm in more detail.

First, two variables are initialised to the empty set, `ASTops`, which will store the inferred AST node operations (line 3), and `matches`, which the algorithm uses to store pairs of matched nodes between the old and new AST (line 4). Then, the traversal path to the common covering nodes is found, and, using this path, the common covering nodes are obtained from both ASTs (lines 5-7).

Next, the algorithm matches outliers, i.e., the nodes that have not been affected by any change. Every descendant node of the common covering node in the old AST is checked against the changes. A change does not affect a node if the code that the node represents is completely before the change, or completely after it. If the offset of the node is before the change, the change does not impact the offset of the node either. However, if the offset of a node is after the offset of the change, it alters the offset of the node. Changing the offset of a node can also be seen as shifting the node to the left or to the right. Text insertions shift the offset by the length of the inserted text, whereas text deletions shift the offset by the opposite number (- length of deleted text). These offset shifts are computed by the function `getChangeOffset` and accumulated in a variable `deltaOffset` (line 12). If no changes affect the old AST node, then the algorithm looks for its matching node in the new AST by using `deltaOffset` (line 14), and the matched pair of nodes is added to matches (line 15).

In the next step, the algorithm matches nodes that are still unmatched but have the same traversal path starting at the root of their ASTs. The algorithm first loops over all the old AST nodes that are descendants of the `oldCoveringNode` and that have not yet been matched (line 18). For each such node, the traversal path to this node is computed and the algorithm attempts to find the node in the new AST on that path (lines 20-21). This pair of nodes is then matched together if the new AST node is also not yet matched (line 22). In case no node can be found, then there cannot be a match.

Next, the algorithm starts generating node operations. For each matched node, the algorithm generates an update operation if the

code contained in the old AST node is different from the code contained in the matching new AST node (lines 25-27). Then, the algorithm generates an update operation for all pairs of corresponding ancestors of the common covering nodes, as explained in Section 2.2.3. Lastly, a delete operation is generated for each unmatched descendant of the common covering node in the old AST (lines 31-33), and an insertion operation is generated for each unmatched descendant of the common covering node in the new AST (lines 34-36).

3 EVALUATION: VISUALISING CHANGES WITH RACKETVIZ

We have evaluated RacketLogger by using the AST changes it captures to create an interactive visualisation of the changes it has logged. To this end, we have built a second plugin for DrRacket, RacketViz. RacketLogger supports registering a callback function which will be called each time a set of node operations is inferred. RacketViz plugs into RacketLogger through this callback. Listing 2 shows how such a callback can be registered. RacketLogger provides five arguments to the callback: the old AST, the new AST, the inferred node operations, the changes connecting both ASTs, and an object which indicates in which tab the changes occurred (in DrRacket, a developer can use multiple tabs simultaneously). Hence, the first four arguments provide all information on the changes, both textually and at the level of AST nodes. We refer back to Figure 3 for illustrative values of the first four arguments.

Listing 2: Registration of the callback function of RacketLogger.

```
(set-AST-inference-callback!
  (lambda (old-ast new-ast inferred-node-ops
      changes-obj defs-text)
    ...))
```

RacketViz implements a visualisation of the current AST of the program, which is updated according to the changes made by the developer whenever the program reaches a parsable state. To this end, RacketViz uses the information on AST node changes provided by RacketLogger, and creates a visualisation that is encoded in the dot language, so that it can be converted into an image by GraphViz [2]. This image is shown to the user in the a frame within the DrRacket editor, and the image is updated every time when AST node operations are inferred. An example image is shown in Figure 6.

Since RacketViz is provided with the inferred AST node operations, it can colour the nodes of the new AST according to these operations: update operations are coloured blue, and insert operations are coloured green. RacketViz does not show the old AST, because this would be cumbersome to do within the small DrRacket frame. Hence, delete operations cannot be visualised, as the deleted nodes are no longer present in the new AST. However, it is entirely possible to also generate images for the old AST, where delete operations could be shown.

Finally, when the programmer changes to another tab, RacketViz loads the image for the corresponding tab. This can simply be retrieved from memory, where RacketViz stores the last generated image for every tab.

Algorithm 1: Inferencing algorithm for AST node operations.

```
1  affects(change, node, offset) returns true if a change affects a node,
     i.e., if the change is made within the boundary of the node.
2  getCommonCoveringPath(oldAST, newAST, changes) returns a list
     describing the path to the common covering node, by finding a local
     covering node in both oldAST and newAST, and extracting their
     common path.
   input  : The old AST, oldAST, the new AST, newAST, and the
            textual changes, changes.
   output : A set of AST node operations, ASTops.
3  ASTops := ∅;                            // AST node operations.
4  matches := ∅;                           // Pairs of matched nodes.
5  coveringPath :=
     getCommonCoveringPath(oldAST, newAST, changes);
6  oldCoveringNode := getNode(oldAST, coveringPath);
7  newCoveringNode := getNode(newAST, coveringPath);
   // Match outliers.
8  foreach oldNode ∈ getDescendants(oldCoveringNode) do
9      deltaOffset := 0;
10     foreach change ∈ changes do
11         if affects(change, oldNode, deltaOffset) then
             Continue foreach line 8 ;
12         else deltaOffset := deltaOffset +
             getChangeOffset(change, oldNode, deltaOffset);
13     end
14     if ∃newNode ∈ getDescendants(newCoveringNode) :
         getOffset(oldNode) + deltaOffset = getOffset(newNode) then
15         matches := matches ∪ (oldNode, newNode);
16     end
17 end
   // Match same-path nodes.
18 foreach oldNode ∈ getDescendants(oldCoveringNode) do
19     if oldNode ∉ getOldNodes(matches) then
20         oldPath := getNodePath(oldNode, oldAST);
21         newNode := getNode(newAST, oldPath);
22         if newNode ≠ null and
             newNode ∉ getNewNodes(matches) then
             matches := matches ∪ (oldNode, newNode) ;
23     end
24 end
   // Infer node operations.
25 foreach (oldNode, newNode) ∈ matches do
26     if getText(oldNode) ≠ getText(newNode) then
         ASTops := ASTops ∪ makeUpdateOp(oldNode, newNode) ;
27 end
28 foreach (oldParent, newParent) ∈
     parentsOnPath(coveringPath, oldAST, newAST) do
29     ASTops := ASTops ∪ makeUpdateOp(oldParent, newParent);
30 end
31 foreach oldNode ∈ getDescendants(oldCoveringNode) do
32     if oldNode ∉ getOldNodes(matches) then
         ASTops := ASTops ∪ makeDeleteOp(oldNode);
33 end
34 foreach newNode ∈ getDescendants(newCoveringNode) do
35     if newNode ∉ getNewNodes(matches) then
         ASTops := ASTops ∪ makeInsertOp(newNode);
36 end
```

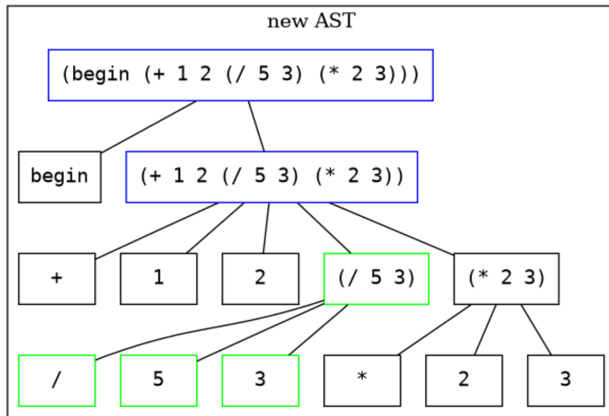

Figure 6: AST visualisation by RacketViz. Node updates are shown in blue, whereas node inserts are shown in green.

4 USER STUDY

To evaluate the usefulness of RacketLogger as a building block for any application that requires information about changes, we have conducted a user study. RacketViz, a tool enabled by RacketLogger, was installed on the computers of five participants, all students in computer science at the bachelor level, using DrRacket daily. Then, they were given about a week to use RacketViz, after which we asked them a series of closed and open questions regarding their experience using RacketViz.

Table 1 lists the closed questions of our user study and the responses of the five participants. The first two questions were used to evaluate the past experience of the participants with DrRacket and Scheme and to know for how much time the participants have used the IDE with RacketViz installed. Next, participants were given a series of statements, for which they had to indicate how much they agreed with each on a scale from 0 to 10.

The three last columns of the table in Table 1 summarise the results of our user study. We see that the participants were already familiar with the DrRacket IDE, having 2 to 4 years of experience using it. The second question asked how intensively they used DrRacket whilst the plugins (RacketViz and RacketLogger as its dependency) were installed. In total, the users have reported using it for 9 hours.

The remaining questions asked about their user experience with RacketViz, where participants rated propositions on a scale from 0 to 10, where 0 indicates a negative user experience, whereas 10 indicates a positive user experience. By looking at all the answers, we see that the users had a positive experience with the plugins. In short, the participants found that DrRacket kept working smoothly, they managed to easily inspect the AST, they found the provided information quite useful and easy to understand, and that the shown AST was quickly updated after changes.

Alongside our closed questions, we also asked the participants some open questions. First, we asked whether any errors occurred whilst using the plugins. Two errors were reported, explaining that the AST is not shown when multiple frames (not tabs) of the DrRacket IDE are open. Second, we asked what else our participants would like to see in the visualisation. We explicitly encouraged wild ideas from our participants. Three participants came up with an idea:

- To add zoom buttons to the AST visualisation.
- To highlight the AST nodes corresponding to run-time errors.
- To highlight the code that was added from one parsable state to the next using a different colour in the editor.

We find that the third idea is particularly interesting since it could make great use of the detailed data about changes provided by RacketLogger. As a third open question, we asked participants whether they had any additional remarks. However, no participant had additional remarks. We plan in the future to extend this preliminary evaluation to evaluate in details the performance, correctness, and usability of RacketLogger and RacketViz.

5 OTHER POTENTIAL APPLICATIONS OF RACKETLOGGER

In this Section, we discuss other applications that could be built on top of the change information provided by RacketLogger. Recall that RacketLogger provides information about textual changes as well as AST changes.

Empirical studies. Developers might be changing complex Scheme expressions more often than straightforward ones. It could also be the case that developers change procedure declarations more often than class declarations. By using the information provided by RacketLogger, and gathering a large and diverse sample of programmers, one could shed light into these and related matters. Declarative change query languages [9, 10] have been developed to facilitate such empirical studies. One could also mine for patterns in the captured changes [5], which can be indicative of refactoring operations for which automated tool support ought to be provided.

Incremental program analysis. Many IDEs already have some form of built-in program analysis to support software development. For example, IntelliJ employs a data flow analysis [4]. When software changes, these analyses have to be rerun to update their results. Clearly, this is a frequent event within an IDE, therefore making it impractical to run a full software analysis upon every change to the code base. As a remedy, incremental program analyses can be used, which update the analysis results based on the changes made to the code [11, Ch. 7],[12, 13]. This makes employing static analysis practical, as an incremental update of the analysis results takes less time than a full analysis of the codebase.

For an incremental analysis to be efficient however, it must have an overview of the changes, allowing the analysis to find the parts of its result that need invalidation and recomputation. In light of this, RacketLogger could be used to provide these changes, and therefore to bring incremental static analysis to DrRacket in the future.

6 RELATED WORK

In this section, we discuss some related work on change logging.

Yoon et al. developed Fluorite [14], an event-logging (or change-logging) plugin for the Eclipse IDE. It logs all low-level events (or changes) that occur in the editor using an XML format. This format

Question	Mean	Min	Max
How familiar are you with DrRacket/Scheme (in years)	3	2	4
How intensively have you used DrRacket after installing RacketViz (in minutes)	108	60	120
Answers on a scale of agreement from 1 to 10			
Does DrRacket works as smoothly as usual when running RacketViz?	9	8	10
Could you easily inspect the AST?	9.6	9	10
Do you find the provided AST information useful?	8.4	8	10
Did the AST shown in DrRacket update quickly when the code was parsable?	9	8	10
Are the inferred node operations clear to you?	8.2	8	9

Table 1: Questions from our user study of RacketViz. The last 5 questions are answered on a scale of agreement from 0 (indicating a negative user experience) to 10 (indicating a positive user experience).

allows the logged textual changes to be used by other plugins. Fluorite only logs textual events, merging changes whenever possible. It does not capture AST changes.

Omori et al. [7] developed OperationRecorder, a change-logging plugin for Eclipse. The goal of this plugin is to more deeply understand code evolution. Traditionally, code evolution is studied by using snapshots in repositories, but the authors found that this traditional way of studying code evolution is incomplete since intermediate changes in the editor are lost. OperationRecorder uses its own inferencing algorithm, that also relies on textual changes.

Negara et al. [6] have developed CodingTracker, a change-logging plugin for Eclipse. Its main goal is to study software evolution in a more complete and precise way. The tool has been used to answer five research questions. One of the questions served as motivation for RacketLogger: "How much code evolution data is not stored using Version Control (VC)?". By performing a study with 15 participants, across 2000 commits and 23002 committed files, they found that on average 37 percent of changes never reach version control. This finding motivates change loggers since they give a more complete picture of code evolution than version control repositories. Negara et al. [6] have also implemented a state-of-the-art AST node operations inferencing algorithm, which we have adapted to Scheme syntax.

Hattori and Lanza have developed Syde, a change-logging plugin for Eclipse. Syde is developed as a tool for collaborative software development. For example, changes are broadcast to all team members of a project, and real-time visualisations of the evolution of the system.

Beller et al. [1] have developed WatchDog, a change-logging plugin for Eclipse, IntelliJ, and Visual Studio Code. By using WatchDog, Beller et al. performed a large-scale study of the habits of developers whilst testing. Their results have shed light on how several activities of developers relate to each other. For example, they found that developers often overestimate the time they spend on testing software.

The most notable difference between existing work and RacketLogger is the compatible IDE. To the best of our knowledge, RacketLogger is the first change-logging plugin for DrRacket. RacketLogger also has common DNA with other change loggers: it logs textual changes [6, 7, 14], merges changes [6, 14], and uses CodingTracker's state-of-the-art AST node operations inferencing algorithm [6].

7 CONCLUSION

In this paper, we have presented RacketLogger, the first change-logging plugin for the DrRacket IDE. We explained that RacketLogger uses a hierarchy of changes, and merges changes, similar to many state-of-the-art change-logging plugins [6], and we discussed AST changes. RacketLogger required adaptation of an existing state inferencing algorithm in order to support Scheme-like languages. We have explained how the nested structure of Scheme code implies node update operations at matching ancestors of the common covering nodes. We have also introduced the challenges that had to be overcome in determining whether an s-expression is affected by textual changes. Finally, we have shown how RacketLogger may be used to build other plugins that require detailed information about changes. To that end, we implemented a change visualiser, RacketViz, which shows the AST of the program as it evolves in a DrRacket frame. We have conducted a preliminary evaluation of RacketViz through a user study, indicating that it was well received by the participants.

REFERENCES

[1] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Trans. Software Eng.*, 45(3):261–284, 2019. doi: 10.1109/TSE.2017.2776152. URL https://doi.org/10.1109/TSE.2017.2776152.

[2] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing, 9th International Symposium, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 483–484. Springer, 2001.

[3] Lile Hattori and Michele Lanza. Syde: a tool for collaborative software development. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010*, pages 235–238. ACM, 2010. doi: 10.1145/1810295.1810339. URL https://doi.org/10.1145/1810295.1810339.

[4] Zarina Kurbatova, Yaroslav Golubev, Vladimir Kovalenko, and Timofey Bryksin. The intellij platform: a framework for building plugins and mining software data. *arXiv preprint arXiv:2110.00141*, 2021.

[5] Tim Molderez, Reinout Stevens, and Coen De Roover. Mining change histories for unknown systematic edits. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR17)*, 2017.

[6] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference*, volume 7313 of *Lecture Notes in Computer Science*, pages 79–103. Springer, 2012. doi: 10.1007/978-3-642-31057-7_5. URL https://doi.org/10.1007/978-3-642-31057-7_5.

[7] Takayuki Omori and Katsuhisa Maruyama. A change-aware development environment by recording editing operations of source code. In Ahmed E. Hassan, Michele Lanza, and Michael W. Godfrey, editors, *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR*

2008, pages 31–34. ACM, 2008. doi: 10.1145/1370750.1370758. URL https://doi.org/10.1145/1370750.1370758.

[8] Turgut Reis Kursun. RacketLogger: Logging changes from the drracket code editor. Bachelor's thesis, Vrije Universiteit Brussel, Brussels, Belgium, 2021.

[9] Reinout Stevens and Coen De Roover. Querying the history of software projects using QwalKeko. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution, Tool Demo Track, (ICSMe14)*, 2014.

[10] Reinout Stevens, Tim Molderez, and Coen De Roover. Querying distilled code changes to extract executable transformations. *Empirical Software Engineering*, 24(1):491–535, 2019.

[11] Tamás Szabó. *Incrementalizing Static Analyses in Datalog*. PhD thesis, Universitätsbibliothek der Johannes Gutenberg-Universität Mainz, 2021.

[12] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: a DSL for the definition of incremental program analyses. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 320–331. ACM, 2016. doi: 10.1145/2970276.2970298. URL https://doi.org/10.1145/2970276.2970298.

[13] Jens Van der Plas, Quentin Stiévenart, Noah Van Es, and Coen De Roover. Incremental Flow Analysis through Computational Dependency Reification. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020*, pages 25–36. IEEE, 2020. doi: 10.1109/SCAM51674.2020.00008. URL https://doi.org/10.1109/SCAM51674.2020.00008.

[14] YoungSeok Yoon and Brad A. Myers. Capturing and analyzing low-level events from the code editor. In Craig Anslow, Shane Markstrum, and Emerson R. Murphy-Hill, editors, *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools, PLATEAU 2011*, pages 25–30. ACM, 2011. doi: 10.1145/2089155.2089163. URL https://doi.org/10.1145/2089155.2089163.