Clipboard: Speculative System Editing with Naming Recommendation

ABSTRACT

Developers often make *systematic edits* that are similar but not identical edits to multiple locations for bug fixes, feature editions, refactoring and new API adaptation. It is challenging to identify all relevant locations and perform consistent changes to different contexts without introducing new bugs.

To help recommend similar systematic edits and complete code transformation automatically, we implement a tool called Clipboard to identify all matched edit locations and perform consistent edits to all code locations. Our approach simplifies the configuration and invocation process for the code transformation via drag-and-drop. With Clipboard, developers define the template of systematic edit by dragging the example code region to a virtual Clipboard before the editing and taking the snapshot again with drag-and-drop action after the editing. Developers can demonstrate one or more examples to generate an *edit recipe* –a reusable template of complex edit operations.

Given an edit recipe, our tool automatically identifies all matched edit locations and applies the transformation to all code locations. Developers can also invoke Clipboard on the fly and our tool will recommend the most suitable recipe based on the incoming edit stream against multiple edit recipes. Before applying the changes to target edit locations, Clipboard analyzes the naming patterns between related variables and recommends a reasonable name for the identifiers in the target code location. Finally, our tool proactively checks the compilation errors, informs developers of the consequences after performing the transformation with the change previews at multiple code locations, and applies the transformation after approved by the developers.

We evaluate our tool with a test suite of 68 exemplar changed methods from Eclipse JDT and SWT.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Program editors*

General Terms

Search-based software engineering, Experimentation

Keywords

Program Transformation, Speculative Analysis, Systematic Edit, Naming Pattern

1. INTRODUCTION

Recent studies show that developers repeat their own mistakes or unknowingly repeat the errors from other in similar but not identical contexts and they are leaning to reuse existed code fragments with adaptive changes for similar programming tasks to save development effort [37]. Tools such as linked editing [46], CloneTracker [8] and Cleman [40] are able to modify multiple code fragments as one yet they are confined to identical changes on clone regions only. Other API migration tools [38, 2, 36] focus on high-level stylized API transformation with adaptation patterns from codebase.

Sydit [27] and LASE [28] match codebase with the given systematic edit template, abstract context-aware transformation from examples, identify similar edit locations, and apply adaptive changes to similar code segments. However, these tools are confined to matching the codebase against a single recipe and perform similar edits to all matching code locations.

Cookbook [18] actively matches the incoming edit stream with multiple edit recipes to recommend the most suitable transformation rule on the fly. Yet Cookbook cannot guarantee a safe transformation without introducing compilation errors because it evaluates the control and data dependence only when it generates the edit recipe but does not check whether the transformation will invoke new compilation errors to the target after transformation.

We proactively apply the transformation to the target code and invoke Eclipse compilation error checker to evaluate the consequence of the transformation before undoing the change and rolling back to the original version. Motivated by the speculative consequence evaluation [34], we inform developers of the preview of the target context after transformation and the compilation errors caused by the transformation before the change is actually applied to the edit location. If the change will invoke any compilation errors, our tool asks for developer's confirm before applying the transformation. Developer can also select one or more edit locations in the recommendation list and review the preview of the transformation with corresponding compilation errors caused by the transformation. After reviewing the systematic edits in the target locations, the developer is able to apply the systematic edit to one or multiple locations, to all the places that will not invoke compilation errors, or to all places without considering compilation errors.

Similar to Cookbook, our tool matches the suitable edit recipe by tracking living edit stream. Yet in Clipboard, when the developer invokes code completion engine in Eclipse and selects an edit recipe from the pop-up menu, our tool provides the preview of the change with potential compilation errors at the area of additional information next to the popup recommendation menu.

To further simplify the process of generating, selecting and invoking the program transformation rule, Clipboard provides an intuitive drag-and-drop approach to demonstrate the snapshots of the example code region before and after the transformation, and applies the transformation by double click or dragging the item of the example (or edit recipe) to the target context. Once the developer drags the code region to the Clipboard, our tool matches similar context based on the similarity of the syntactic structure and recommend the locations that might require similar systematic edit. After the developer specifies the change by dragging the changed code region and dropping it to the new version of the edit recipe in the Clipboard, our tool generates the transformation result of the top ranked edit location and displays the preview of the systematic edit to the developers. Developers can select one or more candidate edit locations and apply the systematic edits to multiple edit locations.

To leverage the comprehension of systematic edits, we record the values of the matched variables and methods between the example and the target location, extract association rules, and synthesize the recommended names for the variables in the target location. For instance, given a new inserted statement `IActionBar actionBar = fContainer.getActionBars()` in the example, our approach extracts the differences from the match pair of variable types `(IActionBar, IServiceLocator)` and the match pair of method calls `(fContainer.getActionBars(), fContainer.getServiceLocator())`, and automatically recommends the name of the new variable `$v` as `serviceLocator` in the statement `IServiceLocator $v = fContainer.getServiceLocator()`.

To evaluate the usability, intuitiveness and effectiveness for this code recommendation system, we conducted an initial user study with 68 exemplar changed method drawn from the version history of Eclipse SWT from prior work [28, 18] . On average, our tool spent Z seconds to generate edit script and perform the transformation to the edit location which does not invoke compilation error, compared to W seconds in manual edit effort without the tool support. Our consequence evaluation correctly addressed X out of Y systematic edits with an accuracy of X%. Compared to the edit version history in the code repository, our approach correctly recommends X out of Y names of the new identifiers. Our result shows that Clipboard is able to speed up the program transformation and minimize the omission errors in program transformation.

In summary, our paper makes the following contributions.

- To simplify the definition, invocation, and configuration process of example-based systematic edits, we allow developers to manipulate the region of systematic edits intuitively via drag-and-drop. Given a list of edit recipes, we match the most suitable systematic edit on the fly and invoke the transformation based on the incoming edit stream.
- We conduct a speculative analysis of the potential compilation errors caused by the transformation and inform the developer of the potential errors along with the preview of the target code region after the systematic edit. Our approach proactively applies the transformation to the target location, evaluates the compilation error invoked by the change, and rollbacks to the original version.
- To increase the comprehension of the auto-generated transformation code, we leverage systematic naming patterns in program differences and synthesize names of the corresponding identifiers in the target edit location.

The rest of the paper is organized as follows. Section 2 places our tool in the context of related works in code recommendation and systematic edit transformation. Section 3 overviews our approach with an motivation example. Section 4 describes the concrete steps for our tools with matching, filtering and consequence evaluation algorithms. Section 5 presents the preliminary result for the UI prototype of the content assist Eclipse Plugin, followed by the preliminary evaluation in Section 6. The next two sections discusses the threats to the validity, discussion and future work for our program transformation recommendation technique. And the paper concludes in Section 9.

2. RELATED WORK

Example-based Editing ClipBoard finds its roots in *programming by demonstration* (PBD), which is also called programming by examples [25]. In PBD, the user demonstrated one or more examples of a program, and the system generalizes the demonstration into a program that can be applied to other examples. Simultaneous editing [29] performs a group of repetitive text editing simultaneously with selection guessing [31] and text constraints [30]. Although the PBD text editors, such as Sublime [17], are able to provide multiple selection and batch editing functions to help developers, they can only make the same textual changes to multiple locations simultaneously. Clipboard exploits program AST structure and performs similar but not identical systematic editing to multiple places.

LIBSYNC [38] and spdiff [1, 2] detects the difference in API usage from multiple instance and migrate the programs by learning API usage adaptation patterns. Yet these API migration tools are confined to stylized API usage and fail to consider dependence constraints of the surrounding context. Our approach supports expressive and customizable transformation to multiple edit locations with customized context-awareness systematic edits to each location.

Semdiff [6] automatically recommends adaptive changes in the face of high-level method additions and deletions, while our approach focuses on more fine-grained systematic transformation and recommend similar edits based on the syntactic structure similarity. Negara et.al [35] mine fine-grained frequent code change patterns with the AST nodes but are unable to transform the similar change to multiple edit locations. PRECISE [48] extracts usage pattern from existing API usage and adaptively recommend parameters based on the current context.

Sydit [27] generates edit scripts from a single systematic edit example and requires developers to specify the target locations before applying the transformation. Whereas LASE [28] overcomes the insufficient transformation rule generation from a single example and identifies the target transformation locations automatically. Different from these tools, Clipboard is able to actively match the incoming edit stream with multiple edit templates to recommend the most suitable transformation rule with compilation error checking.

Code clone management Recent research results pointed out that software system inevitably contain a large amount of similar code due to the copy-and-paste programming practice [5, 19]. These similar or identical code fragments, called *code clones*, may impact software quality with inconsistent modifications made to the cloned code snippets [10]. CP-Miner [24] uses data mining techniques to detects copy-paste related bugs, Dejavu [11] identifies syntactic inconsistency bugs based on the assumption that duplicated code fragments generally intend to remain identical, while SPA [41] facilitates state-control and data-dependence analysis to detect semantic inconsistency in ported code. Different from these clone-related bugs detection tools, ClipBoard focuses on performing semantic consistency transformation with compilation error checking rather than identifying inconsistency changes after the edits are applied to the codebase.

Some clone management tools, such as linked editing [46], Cleman [40] and HotClones [44] detect code clones, allow developers to modify multiple code clones as one, and track changes in separate clone groups. Yet none of these tools are able to apply context-awareness edit transformation. Other clone management tools support live clone change tracking. CCEvents [49] continuously monitors code repositories and provides timely notifications to the stakeholders based on contextual clone events. CloneBoard [7] monitors copy-and paste activities in the Eclipse and offers resolution strategies for inconsistently modified clones, yet it is confined to the clipboard activity in the Eclipse IDE and cannot perform semantic consistence transformations within clone group. CloneTracker [8] applies clone region descriptors to modify multiple sections of code consistently. Yet it fails to perform similar but not identical systematic edits based on the context dependency analysis. Different from these tools for clone-based change management, ClipBoard focuses on the syntactic similar but not identical code regions and performs adaptive changes to multiple edit locations.

Program synthesis and code completion It is common that the programmer knows what type of object he needs, but does not know how to write the code to get the object. Program synthesis tools, such as Jungloid Mining [26] and CodeHint [12], search for a chain of method calls to complete the calling path at runtime based on specifications. SemFix [39] automatically repairs methods based on symbolic execution, constraint solving and program synthesis while MintHint [20] identifies expression based on the repaired pattern and synthesize repair hints from these expressions. GenProg [14] is another automatic program repair tool which uses generic programming to synthesize potential bug-fixes based on test suites. Different from these search based program synthesis tools, our approach focuses on transforming systematic edits with syntactic consistency across multiple edit locations.

GraPacc [36] extracts context features, rank the best matched pattern from the database and fill in the code automatically. Yet these tools are confined to the pre-defined specifications or API usage patterns while in our approach, developers can customize the systematic edit template on the fly before applying the transformation to the target code region. Our prior work, Cookbook [18], automatically matches the incoming edit stream against multiple edit recipes, trying to find the most suitable recipe without requiring users to specify the recipe they prefer to apply. Yet it does not recognize naming patterns between related types and variables, and fails to evaluation the consequence of the transformation based on the control dependency in the target code fragment. ClipBoard overcomes both these limitations of CookBook with a straightforward drag-and-drop script generation process. These interaction process is similar to the Drag-and-drop Refactoring [23], which improves the invocation and completion approach of refactoring in an intuitive manner but is confined to a set of pre-defined refactoring templates. Some code completion tools such as WitchDoctor [43] and BeneFactor [13], detect manual refactoring on the fly and automatically complete the code according to the command from the developers.

Eclipse Content Assist [16] provides couple of auto-completion engines and suggests available methods based on live editing stream. Yet pre-defined transformation templates are required and Eclipse Content Assist does not consider the control and data dependency and thus might inject some errors when performing systematic edit transformation.

Speculative Transformation Codebase Replication [34, 32] maintains a copy of developer's codebase, makes that copy codebase available for offline analyses to run without disturbed by the developer, and implicates the potential offline code changes continuously. Another proactive analysis tool–Quick Fix Scout [33], aims to check the consequence of the quick fix recommendation in Eclipse platform and inform the developers of the potential conflicts after applying the quick fix. It maintains a hidden copy of the source file that is under editing and fore-apply the quick fix separately at the background without affecting the workspace. ClipBoard focuses on systematic edit checking before it is actually applied to the source code, rather than general code changes or quick fix recommendations. Our approach can also proactively check multiple edit locations of the given transformation template.

Different from change impact analysis tools such as Chianti [42] that analyzes the changes between two versions of the applications based on the call graph for regression test selection, we consider control and data dependency when we generate the transformation code not when we check the consequence of the transformation. Our approach proactively applies the transformation to the source code and invokes Eclipse compilation error checker to identify compilation errors before undoing the change to the original version of the codebase.

Our tool is also different from the harmfulness evaluation of the code clone [47], which uses machine learning technique to investigate common characters that indicate potential harmfulness. We recommend developers with compilation errors that will be invoked after applying the transformation in a concrete manner rather than inform them of the high-level harmfulness analysis.

Naming Pattern Lawrie et.al [22] conduct a comprehensive study of identifier names and they find that there is no statistical difference between full words and abbreviations in many cases. Caprile et.al. [4] reconstruct identifier names based on both standard lexicon and standard syntax, yet it is confined to pre-defined naming standards.

Butler et.al. [3] conduct an empirical study of conventional Java class naming patterns and patterns of class names related to inheritance by identifying common grammatical structures of Java class identifiers. Singer et.al. [45] consider semantic information encoded in Java class names and create a prototype to inform programmers of particular problems or optimization opportunities in their code based on the naming patterns. Høst et.al. [15] exploit whether or not a method name and implementation are likely to be good matches for each other and provide a simple pattern-based naming recommendation approach to evaluate reasonable names for the identifiers. Kashiwabara et.al [21] recommend candidate verbs for method names based on the association rules extracted from similar code fragments and help developers consistently use various verbs in method names. Aligned with the idea of extracting association rules from similar code fragments, we construct recommended variable names based on the related contexts of the systematic edits. Our approach is able to generate multiple reasonable names for different edit locations based on the naming patterns.

3. MOTIVATING EXAMPLE

```
    public IActionBars getActionBars() {
        - if (fContainer == null) {
        + IActionBars actionBars = fContainer.getActionBars();
        + if (actionBars == null && !fContainerProvided) {
              return Utilities.findActionBars(fComposite);
        }
        - return fContainer.getActionBars();
        + return actionBars;
    }
}
    public IServiceLocator getServiceLocator() {
        - if (fContainer == null) {
        + IServiceLocator serviceLocator = fContainer.getServiceLocator();
        + if (serviceLocator == null && !fContainerProvided) {
              return Utilities.findSite(fComposite);
        }
        - return fContainer.getServiceLocator();
        + return serviceLocator;
        }
    }
}
```

Figure 1: Motivation Example from org.eclipse.compare.CompareEditorInput between v20061120 and v20061218.

This section overviews the workflow of our tool based on a systematic edit from `org.eclipse.compare.CompareEditor Input` between `v20061120 and v20061218`. Figure 1 displays part of the diff patch of this change, the code in black remains unchanged while the added code is illustrated in blue with `"+"` ahead of line and the deleted statements is in red with `"-"`. The function `getActionBars()` and `getServiceLocator()` in the class `CompareEditorInput` utilize very similar but not identical context with a similar edit as well: removing a condition statement, adding an initialization sentence and add another condition statement afterward. The only difference of this change is the type of the variables. Given this similar yet not identical changes, SyditRecommender first generates an abstract edit transformation recipe for further use.

Later, when developers type in some edits in a similar context, our tool invokes context assist engine in Eclipse and displays the recommendation options for all suitable transformation rules sorted in relevance and popularity order. When developers go through the options in the pop-up menu, our tool provides transformation preview on the target program as well as the consequence of this code change, that is, which errors will be invoked once applying this change. After checking the change preview and corresponding consequence evaluation, developers simply click the recommendation options and perform the edits on the target program.

4. APPROACH

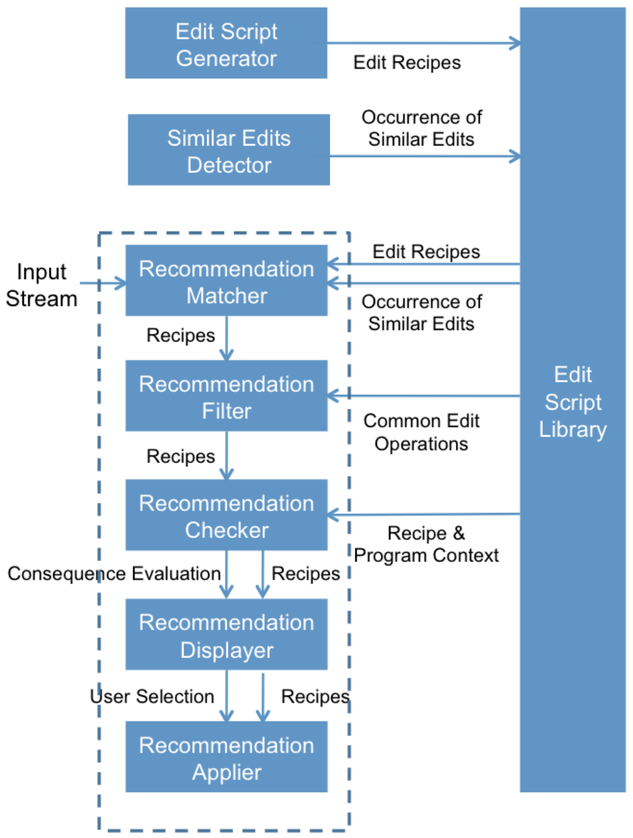


Figure 2: Workflow of SyditRecommender. The boundary of SyditRecommender is illustrated with dotted box

Figure 2 shows the workflow of SyditRecommender. Users can define new edit recipes by demonstrating examples to Recommendation Generator. Our tool applies an AST differencing algorithm to create an edit script and generates the most specific edit scripts. At the same time, our tool automatically investigates all similar edits in package level and stores both the edit recipe as well as the occurrence of the similar systematic changes to the Edit Script Library in the format of XML file. We extend the generation and detection mechanisms of edit scripts in Sydit [27] and LASE [28] so that our tool can locate all similar edits in package level automatically once a new recipe is demonstrated. The information we store in edit script library includes context information, edit operations and corresponding occurrence for each recipe.

SyditRecommender monitors user's editing stream and simultaneously matches the context with all the recipes in the library. Section 4.1 details the matching algorithm while Recommendation Filter in section 4.2 depicts how our tool filters out irrelevant recipes with respect to un-matching edit operations for different the scripts. Section 4.3 describes the mechanism we use to compute the consequence of the systematic transformation.

Finally, we call the content assist engine (also known as code completion engine) in Eclipse to display the ranked candidate recipe list in a pop-up menu regarding the relevance and popularity. And developers can review the potential consequence and the corresponding preview of the selected program transformation rule. After comparing different edit recipes, the developers select the preferred one with a simple click on the menu option and the transformation rule is directly applied to the target program. More details of our tool are described below with respect to how SyditRecommender matches with the context, filters out the irrelevant recipes regarding the editing operations and evaluates the dependency conflicts and compilation errors that might cause after performing the preferred proposal.

4.1 Recommendation Matcher

Algorithm 1 Pseudocode For the Matcher

```
Function {getCandidateScripts} {context, script list}
for all script $\in$ scriptLibrary do
    weight := 0
    weight += forward search the concrete identifiers in the script
    weight += forward search the generic identifiers in the script
    weight += backward search the concrete identifiers in the script
    weight += backward search the generic identifiers in the script
    weight += occurrenceWeight of the script
    if weight > threshold then
        add script to the candidate list
    end if
end for
sort the candidate list based on the matching weight
return candidate list
EndFunction
```

Given the input stream edits, SyditRecommender searches for the method context in each edit recipe in the script library. The goal of the matching algorithm shown in Algorithm 1 is to find the most suitable scripts that matches for the current context in the sense that similar changes are more likely to happen between the code snippets with similar contexts. Aligned with matching technique in [27, 28], we conduct search for the concrete type, method and variable names exactly as it is, as well as the abstract identifiers by its syntax nodes. Here the concrete identifiers indicate concrete class name or keywords. For instance, in the statement `Iterator it = new Iterator()`, the node `=`, `new` and `Iterator` are identical in all examples yet the variable name `it` is different and is abstracted to a generic identifier `v$0` . By overriding the forward completion and backward completion functions in content assist engine, we analyze both upstream dependency and downstream dependency of the context. As a result, we get two lists of suggestions from edit templates. In the next step, we take the occurrence for each script into consideration and calculate the matching weight for each script. We finally rank all the candidate recipes whose matching weight is beyond the threshold.

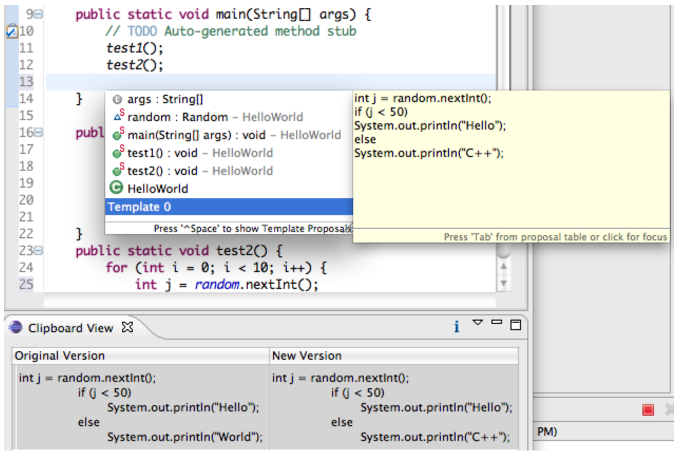
4.2 Recommendation Filter

All the candidate scripts are further elaborated with the edit operations of ***DELETE, INSERT, MOVE, UPDATE*** aligned with [27]. For example, the line 2 in Figure 1 will be abstracted as e_A `= delete(if (container == null )`. SyditRecommender keeps 2 versions of the editing file before and after each keystroke, and collects the editing operation for the edit stream, trying to match it with the candidate scripts. If the user is deleting the statement, then our tool will try to match the edit operations with the type of ***DELETE*** in the candidate edit recipes.

4.3 Recommendation Checker

We leverage the speculative analysis idea in Quick Fix Scount [33] to evaluate the potential consequence after applying the program transformation proposal. We keep a hidden version for the current editing file with synchronous update to the latest version. With this experimental copy, we safely apply the selected edit proposal and evaluate the consequence of this transformation before the transformation is applied to the real source file. We also facilitate the *uodo change* in Eclipse API that rolls back the associated proposal application to recover the separate copy from one transformation rule and switch to the next one.

5. PRELIMINARY RESULT



```
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        test1();
        test2();

    }

    publ

    }
    public static void test2() {
        for (int i = 0; i < 10; i++) {
            int j = random.nextInt();
```

args : String[]
random : Random – HelloWorld
main(String[] args) : void – HelloWorld
test1() : void – HelloWorld
test2() : void – HelloWorld
HelloWorld
Template 0
Press '^Space' to show Template Proposals

```
int j = random.nextInt();
if (j < 50)
System.out.println("Hello");
else
System.out.println("C++");
```

Press 'Tab' from proposal table or click for focus
Clipboard View
PM)

Original Version

```
int j = random.nextInt();
        if (j < 50)
              System.out.println("Hello");
        else
              System.out.println("World");
```

New Version

```
int j = random.nextInt();
        if (j < 50)
              System.out.println("Hello");
        else
              System.out.println("C++");
```

Figure 3: Content Assist Example

We build a UI prototype for the Eclipse Content Assist Plugin. Figure 3 shows a snapshot of the SyditRecommender that recommends the potential edits proposals based on the editing context which has the features listed below:

- It ranks the candidate proposals with respect to its matching of the editing context and returns the sorted candidate recipe list.
- It caches the consequence of the transformation and warns users by displaying all compilation errors after applying the proposal on a hidden copy of the file under editing.

- It updates potential transformation proposals according to the editing stream and updates the consequence evaluation simultaneously.

As shown in Figure 3, SyditRecommender analyzes the editing context against all editing scripts in the library, computing the matching weight by looking forward and backward for both abstract identifiers as well concrete ones. It then ranks the candidate scripts with both matching weight and occurrence weight and invokes content assist engine in Eclipse to display the result in the pop-up menu shown in the graph. Simultaneously, SyditRecommender checks the consequence of each top ranked transformation rules and returns the compilation result for each recipe, so that whenever programer selects one of the proposals, our tool is able to show the preview of the transformation as well as the consequence evaluation aside in yellow comment area correspondingly. After checking the preview and consequence evaluation, developers selects to apply the preferred proposal with a click on the menu option.

6. EVALUATION

6.1 Experiment on Eclipse SWT

We first evaluate the precision and recall changes in terms of different matching threshold.

We evaluate the accuracy and response time of SyditRecommender with 68 exemplar systematic edits from the version history of Eclipse SWT aligned with the data set used in [28]. We generate 28 edit recipes in the scripts library. The results are shown in Table X. We found that SyditRecommender correctly transforms X out of Y systematic edits correctly with a response time of X miniseconds on average.

For the consequence evaluation, our approach correctly covers X compilation errors warning out of Z on average and Y control dependency and data dependency conflicts out of Z.

6.2 User Study

To assess the usability, intuitiveness and efficiency of the SyditRecommender, we conduct a user study in X participants with Y years development experience in Object-oriented language such as Java and C#. We created Z transformation tasks for the study and separate the participants in terms of its experience so that both groups have the same number of people with a similar working experience. One group is asked to use SyditRecommender while the other one preforms the systematic edits without any tool support. X% of participants who use SyditRecommender agree that it is useful and can significantly improve the working efficiency with less human error in program transformation. We compared the time participants spent in two groups and found that the group with SyditRecommender is X seconds faster than the other one, with a X% of improvement in efficiency. And the programs generated by SyditRecommender is Q% similar to the corresponding human editing.

7. THREATS TO VALIDITY

Internal Validity The experience difference in two group could have affected the response time and result of the evaluation.

External Validity All the participants are graduate or undergraduate students in University of Texas at Austin major in Computer Science or Software Engineering. Although the participants have diverse experiences in Java, refactoring and Eclipse IDE, they might not be representative of all software developers in industry. Our recommendation tool is also confined with the Eclipse and supports Java only.

8. DISCUSSION AND FUTURE WORK

9. CONCLUSION

In this paper, we present the preliminary result of our IDE recommendation for systematic edits using edit recipes learned from examples. We describe our approach to match, filter, evaluate the consequence of the program transformation and display the candidate proposal list with content assist engine in Eclipse with preview and consequence preview aside. Our tool eliminates the tedious invoking and configuration process for the program transformation with a safe consequence evaluation of potential compilation errors for the systematic change. Our experiment on the Eclipse SWT project illustrates that SyditRecommender is able to detect and apply the similar but not identical systematic edits with an accuracy of X%. The user study we conduct prove that SyditRecommender is more intuitive than the traditional program transformation tools with a X% efficiency improvement on average.

10. REFERENCES

[1] J. Andersen and J. L.Lawall. Generic patch inference. *ASE 2008*, pages 337–346, 2008.

[2] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo. Semantic patch inference. *ASE 2012*, pages 382–385, 2012.

[3] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Mining java class naming conventions. In *ICSM*, pages 93–102, 2011.

[4] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM*, pages 97–107, 2000.

[5] J. R. Cordy. Exploring large-scale system similarity using incremental clone detection and live scatterplots. *ICPC 2011*, pages 151–160, 2011.

[6] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE*, pages 481–490, 2008.

[7] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *ICSM*, pages 169–178, 2009.

[8] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE*, pages 158–167, 2007.

[9] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[10] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *OOPSLA*, pages 175–190, 2010.

[11] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *OOPSLA*, pages 175–190, 2010.

[12] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *ICSE*, 2014.

[13] X. Ge, Q. L.DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. *ICSE 2012*, pages 211–221, 2012.

[14] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[15] E. W. Høst and B. M. Østvold. Debugging method names. In *ECOOP*, pages 294–317, 2009.

[16] http://www.eclipse.org/recommenders/manual/.

[17] http://www.sublimetext.com/.

[18] J. Jacobellis, N. Meng, and M. Kim. Cookbook: In situ code completion using edit recipes learned from examples. 2014.

[19] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC/SIGSOFT FSE*, pages 55–64, 2007.

[20] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: Automated synthesis of repair hints. *CoRR*, abs/1306.1286, 2013.

[21] Y. Kashiwabara, Y. Onizuka, T. Ishio, Y. Hayase, T. Yamamoto, and K. Inoue. Recommending verbs for rename method using association rule mining. In *CSMR-WCRE*, pages 323–327, 2014.

[22] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *ICPC*, pages 3–12, 2006.

[23] Y. Y. Lee, N. Chen, and R. E.Johnson. Drag-and-drop refactoring: Intuitive and efficient program transformation. *ICSE 2013*, pages 23–32, 2013.

[24] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.

[25] H. Liberman. Your wish is my command: Programming by example. In *Morgan Kaufmann Publisher*, 2001.

[26] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61, 2005.

[27] N. Meng, M. Kim, and K. S.McKinley. Systematic editing: Generating program transformations from an example. *PLDI 2011*, pages 329–342, 2011.

[28] N. Meng, M. Kim, and K. S.McKinley. Lase: Locating and applying systematic edits by learning from examples. *ICSE 2013*, pages 502–511, 2013.

[29] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference, General Track*, pages 161–174, 2001.

[30] R. C. Miller and B. A. Myers. Lapis: smart editing with text structure. In *CHI Extended Abstracts*, pages 496–497, 2002.

[31] R. C. Miller and B. A. Myers. Multiple selections in smart text editing. In *IUI*, pages 103–110, 2002.

[32] K. Muslu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Improving ide recommendations by considering global implications of existing recommendations. *ICSE 2012*, pages 1349–1352, 2012.

[33] K. Muslu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. *OOPSLA 2012*, pages 669–682, 2012.

[34] K. Muslu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Making offline analyses continuous. *FSE 2013*, pages 323–333, 2013.

[35] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *ICSE*, 2014.

[36] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N.Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. *ICSE 2012*, pages 69–79, 2012.

[37] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. *ASE 2013*, 2013.

[38] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. *OOPSLA 2010*, pages 302–321, 2010.

[39] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *ICSE*, pages 772–781, 2013.

[40] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Cleman: Comprehensive clone group evolution management. In *ASE*, pages 451–454, 2008.

[41] B. Ray, M. Kim, S. Person, and N. Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *ASE*, pages 367–377, 2013.

[42] X. Ren, B. G. Ryder, M. Störzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *ICSE*, pages 664–665, 2005.

[43] S. R.Foster, W. G.Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. *ICSE 2012*, pages 222–232, 2012.

[44] N. Schwarz. Hot clones: Combining search-driven development, clone management, and code provenance. In *ICSE*, pages 1628–1629, 2012.

[45] J. Singer and C. C. Kirkham. Exploiting the correspondence between micro patterns and class names. In *SCAM*, pages 67–76, 2008.

[46] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VL/HCC*, pages 173–180, 2004.

[47] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can i clone this piece of code here? In *ASE*, pages 170–179, 2012.

[48] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *ICSE*, pages 826–836, 2012.

[49] G. Zhang, X. Peng, Z. Xing, S. Jiang, H. Wang, and W. Zhao. Towards contextual and on-demand code clone management by continuous monitoring. In *ASE*, pages 497–507, 2013.